An Introduction to

AD MODEL BUILDER Version 4

For Use in Nonlinear Modeling and Statistics

Otter Research Ltd.
Box 2040,
Sidney B.C. V8L 3S3
CANADA

Telephone: Voice or FAX 250–655–3364
Email:otter@otter-rsch.com

Contents

Chapter 1

Getting started with AD Model Builder

This manual describes AD Model Builder, the fastest, most powerful software for rapid development and fitting of general nonlinear statistical models available. The accompanying demonstration disk has a number of example programs from various fields including chemical engineering, natural resource modeling, and financial modeling. As you will see, with a few statements you can build powerful programs to solve problems that would completely defeat other modeling environments. The AD Model Builder environment makes it simple to deal with recurring difficulties in nonlinear modeling, such as restricting the values which parameters can assume, carrying out the optimization in a stepwise manner, and producing a report of the estimates of the standard deviations of the parameter estimates. And these techniques scale up to models with at least 1500 independent parameters on a 120 MH Pentium PC and more on more powerful platforms. So, if you are interested in a really powerful environment for nonlinear modeling – read on!

AD Model Builder provides a template-like approach to code generation. Instead of needing to write all the code for the model the user can employ any ASCII file editor to simply fill in the template, describing the particular aspects of the model – data, model parameters, and the fitting criterion to be used. With this approach the specification of the model is reduced to the absolute minimum number of statements. Reasonable default behaviour for various aspects of modeling such as the input of data and initial parameters and reporting of results are provided. Of course it is possible to override this default behaviour to customize an application when desired. The command line argument -ind NAME followed by the string NAME changes the default data input file to NAME.

The various concepts embodied in AD Model Builder are introduced in a series of examples. You should at least skim through each of the examples in the order they appear so that you will be familiar with the concepts used in the later examples. The examples disk contains the AD Model Builder template code, the C++ code produced by AD Model Builder and the executable programs produced by compiling the C++ code. This process of producing the executable is automated so that the user who doesn't wish to consider the vagaries of C++

programming can go from the AD Model Builder template to the compiled executable in one step. Assuming that the C++ compiler and AD Model Builder and AUTODIF libraries have been properly installed, then to produce a AD Model Builder executable it is only necessary to type `makeadm root` where `root.tpl` is the name of the ASCII file containing the template specification. To simplify model development two modes of operation are provided, a safe mode with bounds checking on all array objects and an optimized mode for fastest execution.

AD Model Builder achieves its high performance levels by employing the AUTODIF C++ class library. AUTODIF combines an array language with the reverse mode of Automatic differentiation supplemented with precompiled adjoint code for the derivatives of common array and matrix operations. However, all of this is completely transparent to the AD Model Builder user. It is only necessary to provide a simple description of the statistical model desired and the entire process of fitting the model to data and reporting the results is taken care of automatically.

Although C++ potentially provides good support for mathematical modeling, the language is rather complex – it cannot be learned in a few days. Moreover many features of the language are not needed for mathematical modeling. A novice user who wishes to build mathematical models may have a difficult time deciding which features of the language to learn and which features can be ignored until later. AD Model Builder is intended to help overcome these difficulties and to speed up model development. When using AD Model Builder most of the aspects of C++ programming are hidden from the user. In fact the beginning user can be almost unaware that C++ underlies the implementation of AD Model Builder. It is only necessary to be familiar with some of the simpler aspects of C or C++ syntax.

To interpret the results of the statistical analysis AD Model Builder provides simple methods for calculating the profile likelihood and Markov chain simulation estimates of the posterior distribution for parameters of interest (Hastings-Metropolis algorithm).

A short description of each example follows.

A very simple example. This is a trivial least squares linear model included simply to introduce the basics of AD Model Builder.

A simple nonlinear regression model for estimating the parameters describing a von Bertalanffy growth curve from size-at-age data. AD Model Builder's robust regression routine is introduced and used to illustrate how problems caused by "outliers" in the data can be avoided.

A chemical kinetics problem. A model defined by a system of ordinary differential equations. The purpose is to estimate the parameters which describe the chemical reaction.

A problem in financial modeling. A Generalized Autoregressive Conditional Heteroskedasticity or GARCH model is used to attempt to describe the time series of returns from some market instrument.

A problem in natural resource management. The Schaeffer-Pella-Tomlinson Model for investigating the response of an exploited fish population is developed and extended to include a Bayesian times series treatment of time-varying carrying capacity. This example is interesting because the model is rather tempermental and several techniques for producing reliable convergence of the estimation procedure to the correct answer are described. For one of the data sets over 100 parameters are estimated.

A Simple Fisheries catch-at-age model. These models are used to try and estimate the exploitation rates etc. in exploited fish populations.

More complex examples are presented in subsequent chapters.

An AD Model Builder template consists of up to nine sections. Six of these sections are optional. Optional sections are enclosed in brackets [ ]. The optional FUNCTION keyword defines a subsection of the `PROCEDURE_SECTION`.

```
DATA_SECTION

[INITIALIZATION_SECTION]

PARAMETER_SECTION

[PRELIMINARY_CALCS_SECTION]



PROCEDURE_SECTION

[FUNCTION]

[REPORT_SECTION]

[RUNTIME_SECTION]

[TOP_OF_MAIN_SECTION]

[GLOBALS_SECTION]

[BETWEEN_PHASES_SECTION]

[FINAL_SECTION]
```

The simplest model contains only the three required sections, a `DATA_SECTION`, a `PARAMETER_SECTION`, and a `PROCEDURE_SECTION`.

To illustrate the method we begin with a simple statistical model which is to estimate the parameters of a linear relationship of the form

$$Y_i = ax_i + b \qquad \text{for } 1 <= i <= n$$

where x_i and Y_i are vectors, and a and b are the model parameters which are to be estimated. The parameters are estimated by the method of least-squares that is we find the values of

a and b so that the sum of the squared differences between the observed values Y_i and the predicted values $ax_i + b$ is minimized. That is we want to solve the problem

$$\min_{a,b} \sum_{i=1}^{n} (Y_i - ax_i - b)^2$$

The template for this model is in the file SIMPLE.TPL. To make the model one would type `makeadm simple`. The resulting executable for the model is in the file SIMPLE.EXE. The contents of SIMPLE.TPL are. (Anything following // is a comment.)

```
DATA_SECTION
  init_int nobs             // nobs is the number of observations
  init_vector Y(1,nobs)     // the observed Y values
  init_vector x(1,nobs)
PARAMETER_SECTION
  init_number a
  init_number b
  vector pred_Y(1,nobs)
  objective_function_value f
PROCEDURE_SECTION
  pred_Y=a*x+b;       // calculate the predicted Y values
  f=regression(Y,pred_Y);  // do the regression -- the vector of
                           // observations goes first
```

The main requirement is that all keywords must begin in column 1 while the code itself must be indented.

Roughly speaking, the data consist of the stuff in the real world which you observe and want to analyze. The data section describes the structure of the data in your model. Data objects consist of integers (`int`) and floating point numbers(`number`), and these can be grouped into one dimensional (`ivector` and `vector`) and two dimensional (`imatrix` and `matrix`) arrays. The "`i`" in `ivector` distinguishes a vector of type `int` from a vector of type `number`. For arrays of type `number` there are currently arrays up to dimension 7.

Some of your data must be read in from somewhere, that is, you need to start with something. These data objects are referred to as initial objects and are distinguished by the prefix `init`, such as `init_int` or `init_number`. All objects prefaced with `init` in the `DATA_SECTION` are read in from a data file in the order in which they are declared. The default file names for various files are derived from the name of the executable program. If the executable file is named ROOT.EXE then the default input data file name is ROOT.DAT. For this example the executable file is named SIMPLE.EXE so the default data file is SIMPLE.DAT. Notice that once an object has been read in, its value is available to be used to describe other data objects. In this case the value of `nobs` can be used to define the size of the vectors `Y` and `x`. The next line `init_vector Y(1,nobs)` defines an initial vector object `Y` whose minimum valid index is 1, and whose maximum valid index is `nobs`. This vector object will be read in next from the data file. The contents of the file SIMPLE.DAT are shown below.

```
# number of observations
    10
# observed Y values
   1.4  4.7  5.1  8.3  9.0  14.5  14.0  13.4  19.2  18
# observed x values
   -1  0 1  2  3  4  5  6  7  8
```

It is possible to put comment lines in the data files. Comment lines must have the character `#` in the first column.

It is often useful to have data objects which are not initial. Such objects have their values calculated from the values of initial data objects. Examples of the use of non initial data objects are given below.

It is the parameters of your model which provide the analysis of the data (or perhaps more correctly is the values of these parameters as picked by the fitting criterion for the model which provide the analysis of the data). The `PARAMETER_SECTION` is used to describe the structure of the parameters in your model. The description of the model parameters is similar to that used for the data in the `DATA_SECTION`.

All parameters are floating point numbers (or arrays of floating point numbers.) The statement `init_number b` defines a floating point number (actually a double). The preface `init` means that this is an initial parameter. Initial parameters have two properties which distinguish them from other model parameters. First, all of the other model parameters are calculated from the initial parameters. This means that in order to calculate the values of the model parameters it is first necessary to have values for the initial parameters. A major difference between initial data objects (which must be read in from a data file) and initial parameters is that since parameters are estimated in the model it is possible to assign initial default values to them.

The default file name for the file which contains initial values for the initial model parameters is `ROOT.PIN`. If no file named `ROOT.PIN` is found, default values are supplied for the initial parameters. (Methods for changing the default values for initial parameters are described below.) The statement `vector pred_Y(1,nobs)` defines a vector of parameters. Since it is not prefaced with `init` the values for this vector will not be read in from a file or given default values. It is expected that the value of the elements of this vector will be calculated in terms of other parameters.

The statement `objective_function_value f` defines a floating point number (again actually a double). It will hold the value of the fitting criterion. The parameters of the model are chosen so that this value is minimized[1]. Every AD Model Builder template must include a declaration of an object of type `objective_function_value` and this object must be set equal to a fitting criterion. (Don't worry, for many models the fitting criterion is provided for you as in the `regression` and `robust_regression` fitting criterion functions in the current and next examples.

[1]Thus it should be set equal to minus the log-likelihood function if that criterion is used

The PROCEDURE_SECTION contains the actual model calculations. This section contains C++ code and C++ syntax must be obeyed. (Those familiar with C or C++ will notice that the usual methods for defining and ending a function are not necessary and in fact can not be used for the routine in the main part of this section.)

Statements must end with a ";" exactly as with C or C++. The ";" is optional in the DATA_SECTION and the PARAMETER_SECTION. The code uses AUTODIF's vector operations which enable you to avoid writing a lot of code for loops. In the statement pred_Y=a*x+b; the symbol a*x forms the product of the number a and the components of the vector x while +b adds the value of the number b to this product so that pred_Y has the components $ax_i + b$. In the line f=regression(Y,pred_Y); the function regression calculates the log-likelihood function for the regression and assigns this value to the object f which is of type objective_function_value. This code generalizes immediately to nonlinear regression models and can be trivially modified (with the addition of one word) to perform the robust nonlinear regression discussed in the second example. For the reader who want to know, the form of the regression function is described in the Appendix.

Note that the vector of observed values goes first. The use of the regression function makes the purpose of the calculations clearer, and it prepares the way for modifying the routine to use AD Model Builder's robust regression function.

NOTE: The use of LOCAL_CALCS and its variants in the DATA_SECTION and the PROCEDURE_SECTION has greatly reduced the need for the PRELIMINARY_CALCS_SECTION.

The PRELIMINARY_CALCS_SECTION as its name implies permits one to do preliminary calculations with the data before getting into the model proper. Often the input data are not in a convenient form for doing the analysis and one wants to carry out some calculations with the input data to put them in a more convenient form. Suppose that the input data for the simple regression model are in the form

```
# number of observations
     10
# observed Y values  observed x values
    1.4                 -1
    4.7                  0
    5.1                  1
    8.3                  2
    9.0                  3
    14.5                 4
    14.0                 5
    13.4                 6
    19.2                 7
    18                   8
```

The problem is that the data are in pairs in the form (Y_i, x_i), so that we can't read in either the x_i or Y_i first. To read in the data in this format we will define a matrix with nobs rows and 2 columns. The DATA_SECTION becomes

```
DATA_SECTION
  init_int nobs
  init_matrix Obs(1,nobs,1,2)
  vector Y(1,nobs)
  vector x(1,nobs)
```

Notice that since we do not want to read in `Y` or `x` these objects are no longer initial objects, that is their declarations are no longer prefaced with `int`. The observations will be read into the initial matrix object `Obs` so that `Y` is in the first column of `Obs` while `x` is in the second column. If we don't want to change the rest of the code the next problem is to get the first column of `Obs` into `Y` and the second column of `Obs` into `x`. The following code in the `PRELIMINARY_CALCS_SECTION` will accomplish this objective. It uses the function `column` which extracts a column from a matrix object so that it can be put into a vector object.

```
PRELIMINARY_CALCS_SECTION
 Y=column(Obs,1); // extract the first column
 x=column(Obs,2); // extract the second column
```

This section can be skipped on first reading.

To accomplish the column-wise extraction presented above you would have to know that AUTODIF provides the `column` operation. What if you didn't know that and don't feel like reading the manual yet? For those who are familiar with C it is generally possible to use lower level "C-like" operations to accomplish the same objective as AUTODIF's array and matrix operations. In this case the columns of the matrix `Obs` can also be copied to the vectors `x` and `Y` by using a standard `for loop` and the following element-wise operations

```
PRELIMINARY_CALCS_SECTION
  for (int i=1;i<=nobs;i++)
  {
    Y[i]=Obs[i][1];
    x[i]=Obs[i][2];
  }
```

Incidentally, the C-like operation `[]` was used for indexing members of arrays. AD Model Builder also supports the use of `()` so that the above code could be written as

```
PRELIMINARY_CALCS_SECTION
  for (int i=1;i<=nobs;i++)
  {
    Y(i)=Obs(i,1);
    x(i)=Obs(i,2);
  }
```

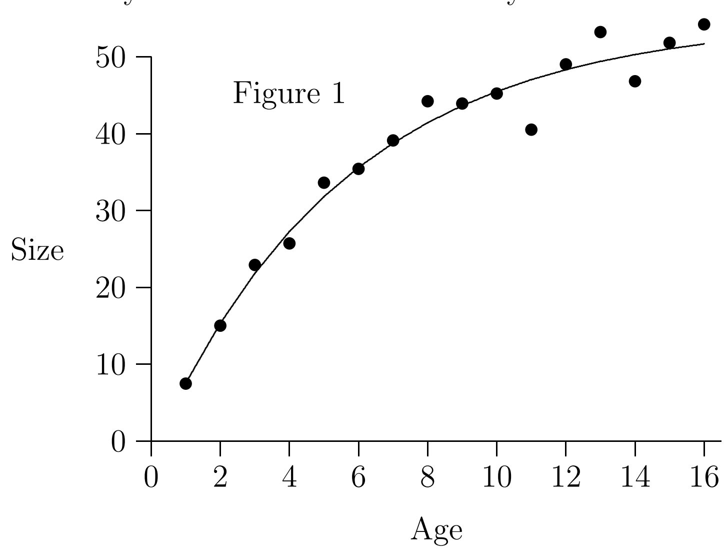
which may be more readable for some users. Notice that it is also possible to define `C` objects like the object of type `int i` used as the index for the `for loop` "on the fly" in the `PRELIMINARY_CALCS_SECTION` or the `PROCEDURE_SECTION`.

By default AD Model Builder produces three or more files `ROOT.PAR` which contains the parameter estimates in ASCII format, `ROOT.BAR` which is the parameter estimates in a binary file format, and `ROOT.COR` which contains the estimated standard deviations and correlations of the parameter estimates. The template code for the simple model is in the file file `SIMPLE.TPL`. The input data is in the file `SIMPLE.DAT`. The parameter estimates are in the file `SIMPLE.PAR`. By default the standard deviations and the correlation matrix for the model parameters are estimated. They are in the file `SIMPLE.COR`

```
index      value      std dev       1       2
   1   a  1.9091e+00 1.5547e-01       1
   2   b  4.0782e+00 7.0394e-01  -0.773       1
```

The format of the standard deviations report is to give the name of the parameter followed by its value and standard deviation. After that the correlation matrix for the parameters is given.

The code for the admodel template for this example is found in the file `VONB.TPL`. This example is intended to demonstrate the advantages of using AD Model Builder's robust regression routine over standard nonlinear least square regression procedures. Further discussion about the underlying theory can be found in the AUTODIF User's Manual, but it is not necessary to understand the theory to make use of the procedure.



Results for nonlinear regression with good data set

```
index          value      std dev       1       2       3
   1   Linf  5.4861e+01 2.4704e+00  1.0000
   2   K     1.7985e-01 2.7127e-02 -0.9191  1.0000
   3   t0    1.8031e-01 2.9549e-01 -0.5856  0.7821  1.0000
```

This example estimates the parameters describing a growth curve from a set of data consisting of ages and size-at-age data. The form of the (von Bertalanffy) growth curve is assumed to be

$$s(a) = L_\infty(\, 1 - \,\exp(-K(a - t_0))\,) \tag{1.1}$$

The three parameters of the curve to be estimated are L_∞, K, and t_0.

Let O_i and a_i be the observed size and age of the i'th animal. The predicted size $s(a_i)$ is given by equation 1.1. The least squares estimates for the parameters are found by minimizing

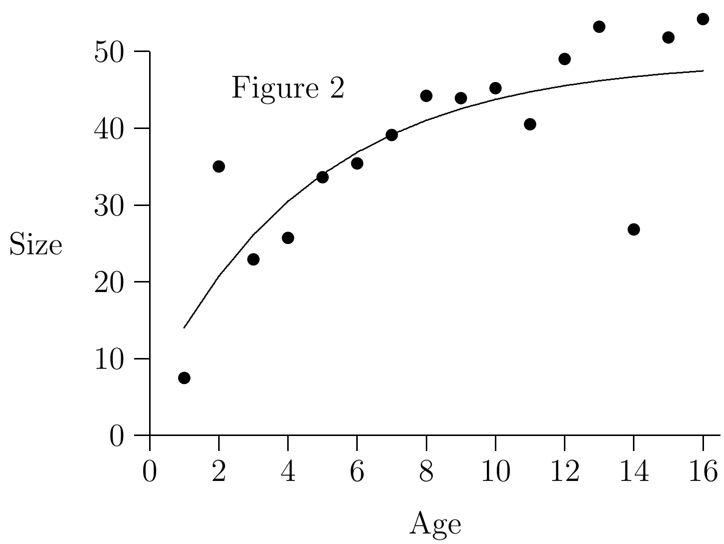
$$\min_{L_\infty,K,t_0} \sum_i \left(O_i - s(a_i)\,\right)^2$$

```
DATA_SECTION
  init_int nobs;
  init_matrix data(1,nobs,1,2)
  vector age(1,nobs);
  vector size(1,nobs);
PARAMETER_SECTION
  init_number Linf;
  init_number K;
  init_number t0;
  vector pred_size(1,nobs)
  objective_function_value f;
PRELIMINARY_CALCS_SECTION
  // get the data out of the columns of the data matrix
  age=column(data,1);
  size=column(data,2);
  Linf=1.1*max(size);  // set Linf to 1.1 times the longest observed length
PROCEDURE_SECTION
  pred_size=Linf*(1.-exp(-K*(age-t0)));
  f=regression(size,pred_size);
```

Notice the use of the `regression function` which calculates the log-likelihood function of the nonlinear least-squares regression. This part of the code is formally identical to the code for the linear regression problem in the simple example even though we are now doing nonlinear regression. A graph of the least-square estimated growth curve and the observed data is given in figure 1. The parameter estimates and their estimated standard deviations which are produced by AD Model Builder are also given. For example the estimate for L_∞ is 54.86 with a standard deviation of 2.47. Since a 95% confidence limit is about $\pm$ two standard deviations the usual 95% confidence limit of L_∞ for this analysis would be 54.86 ± 4.94.

A disadvantage of least squares regression is the sensitivity of the estimates to a few "bad" data points or outliers. Figure 2 show the least squares estimates when the observed size for age 2 and age 14 have been moved off the curve. There has been a rather large change in some of the parameter estimates. For example the estimate for L_∞ has changed from 54.86 to 48.91 and the estimated standard deviation for this parameter has increased to 5.99. This is a common effect of outliers on least-squares estimates. They greatly increase the size of the estimates of the standard deviations. As a result the confidence limits for the parameters are increased. In this case the 95% confidence limits for L_∞ have been increased from 54.86 ± 4.94 to 48.91 ± 11.98.

Of course for this simple example it could be argued that a visual examination of the residuals would identify the outliers so that they could be removed. This is true, but in larger nonlinear models it is often not possible or convenient to identify and remove all the outliers in this fashion. Also the process of removing "inconvenient" observations from data can be uncomfortably close to "cooking" the data in order to obtain the desired result from the analysis. An alternative approach which avoids these difficulties is to employ AD Model Builder's robust regression procedure which removes the undue influence of outlying points without the need to expressly remove them from the data.



Nonlinear regression with bad data set

```
Nonlinear regression with bad data set
index         value      std dev       1       2       3
   1   Linf  4.8905e+01 5.9938e+00  1.0000
   2   K     2.1246e-01 1.2076e-01 -0.8923  1.0000
   3   t0   -5.9153e-01 1.4006e+00 -0.6548  0.8707  1.0000
```

To invoke the robust regression procedure it is necessary to make three changes to the existing code. The template for the robust regression version of the model can be found in the file `VONBR.TPL`.

```
DATA_SECTION
  init_int nobs;
  init_matrix data(1,nobs,1,2)
  vector age(1,nobs)
  vector size(1,nobs)
PARAMETER_SECTION
  init_number Linf
  init_number K
  init_number t0
  vector pred_size(1,nobs)
  objective_function_value f
  init_bounded_number a(0.0,0.7,2)
PRELIMINARY_CALCS_SECTION
  // get the data out of the columns of the data matrix
  age=column(data,1);
  size=column(data,2);
  Linf=1.1*max(size);  // set Linf to 1.1 times the longest observed length
  a=0.7;
PROCEDURE_SECTION
  pred_size=Linf*(1.-exp(-K*(age-t0)));
  f=robust_regression(size,pred_size,a);
```

The main modification to the model involves the addition of the parameter `a`, which is used to estimate the amount of contamination by outliers. This parameter is declared in the `PARAMETER_SECTION`.

```
init_bounded_number a(0.0,0.7,2)
```

This introduces two concepts, putting bounds on the values which initial parameters can take on and carrying out the minimization in a number of stages. The value of `a` should be restricted to lie between 0.0 and 0.7 (See the discussion on robust regression in the AUTODIF user's manual if you want to know where the 0.0 and 0.7 come from). This is accomplished by declaring `a` to be of type `init_bounded_number`. In general it is not possible to estimate the parameter `a` determining the amount of contamination by outliers until the other parameters of the model have been "almost" estimated, that is, until we have done a preliminary fit of the model. This is a common situation in nonlinear modeling and is discussed further in some later examples. So we want to carry out the minimization in two phases. During the first phase `a` should be held constant. In general for any initial parameter the last number in its declaration, if present, determines the number of the phase in which that parameter becomes active. If no number is given the parameter becomes active in phase 1. (Note: For an `init_bounded_number` the upper and lower bounds must be given so the declaration

```
init_bounded_number a(0.0,0.7)
```

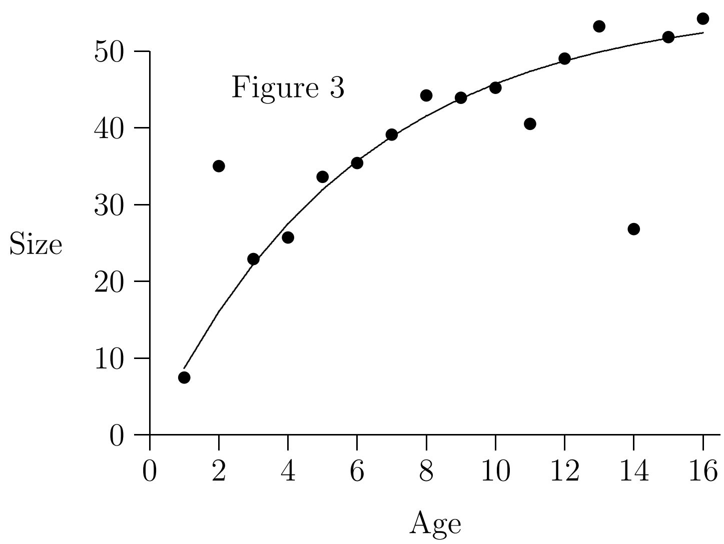
would use the default phase 1. The `2` in the declaration for `a` causes `a` to be constant until the second phase of the minimization. The second change to the model involves the default initial value `a`. The default value for a bounded number is the average of the upper and lower bounds. For `a` this would be 0.35 which is too small. We want to use the upper bound of 0.7. This is done by adding the line

```
a=0.7;
```

in the `PRELIMINARY_CALCS_SECTION`. Finally we modify the statement in the `PROCEDURE_SECTION` including the `regression` function to

```
f=robust_regression(size,pred_size,a);
```

to invoke the robust regression function. That's all there is to it! These three changes will convert any AD Model builder template from a nonlinear regression model to a robust nonlinear regression model.

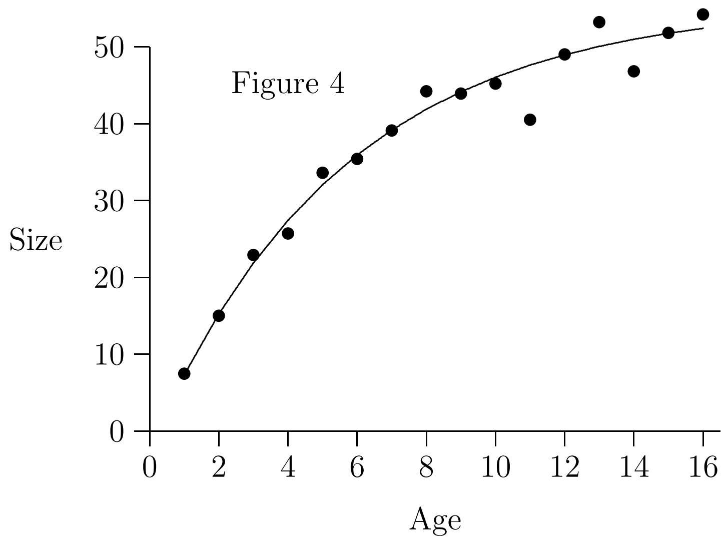


Robust Nonlinear regression with bad data set

```
index          value      std dev       1        2        3        4
   1   Linf  5.6184e+01 3.6796e+00  1.0000
   2   K     1.6818e-01 3.4527e-02 -0.9173  1.0000
   3   t0    6.5129e-04 4.5620e-01 -0.5483  0.7724  1.0000
   4   a     3.6144e-01 1.0721e-01 -0.1946  0.0367 -0.2095  1.0000
```

The results for the robust regression fit to the bad data set are shown in figure 3. The estimate for L_∞ is 56.18 with a standard deviation of 3.68 to give a 95% confidence interval of about 56.18 ± 7.36. Both the parameter estimates and the confidence limits are much less affected by the outliers for the robust regression estimates than they are for the least squares estimates. The parameter `a` is estimated to be equal to 0.36 which indicates that the robust procedure has detected some moderately large outliers.

The results for the robust regression fit to the good data set are shown in figure 4. The estimates are almost identical to the least-square estimates for the same data. This is a property of the robust estimates. They do almost as well as the least-square estimates when the assumption of normally distributed errors in the statistical model is satisfied exactly, and they do much better than least square estimates in the presence of moderate or large outliers. You can lose only a little and you stand to gain a lot by using these estimators.



Robust Nonlinear regression with good data set

```
index          value      std dev       1        2       3       4
   1   Linf  5.5707e+01 1.9178e+00  1.0000
   2   K     1.7896e-01 1.9697e-02 -0.9148  1.0000
   3   t0    2.1490e-01 2.0931e-01 -0.5604  0.7680  1.0000
   4   a     7.0000e-01 3.2246e-05 -0.0001  0.0000 -0.0000  1.0000
```

This example may strike you as being fairly complicated. If so, you should compare it with the original solution using the so-called sensitivity equations. The reference is Bard, *Nonlinear Parameter Estimation*, chapter 8. We consider the chemical kinetics problem introduced on page 233. This is a model defined by a first order system of two ordinary differential equations.

$$\begin{aligned} ds_1/dt &= -\theta_1 \exp(-\theta_2/T)(s_1 - e^{-1000/T} s_2^2)/(1+\theta_3 \exp(-\theta_4/T)s_1)^2 \\ ds_2/dt &= 2\theta_1 \exp(-\theta_2/T)(s_1 - e^{-1000/T} s_2^2)/(1+\theta_3 \exp(-\theta_4/T)s_1)^2 \end{aligned} \tag{1.2}$$

The differential equations describe the evolution over time of the concentrations of the two reactants, s_1, and s_2. There are ten initial parameters in the model, $\theta_1, \ldots, \theta_{10}$. T is the temperature at which the reaction takes place. To integrate the system of differential equations we require the initial concentrations of the reactants, $s_1(0)$ and $s_2(0)$ at time 0.

The reaction was carried out three times at temperatures of 200, 400, and 600 degrees. For the first run there were initially equal concentrations of the two reactants. The second run initially consisted of only the first reactant, and the third run initially consisted of only the second reactant. The initial concentrations of the reactants are known only approximately. They are

Run 1	$s_1(0) = \theta_5 = 1 \pm 0.05$	$s_2(0) = \theta_6 = 1 \pm 0.05$
Run 2	$s_1(0) = \theta_7 = 1 \pm 0.05$	$s_2(0) = 0$
Run 3	$s_1(0) = 0$	$s_2(0) = \theta_8 = 1 \pm 0.05$

The unknown initial concentrations are treated as parameters to be estimated with Bayesian prior distributions on them reflecting the level of certainty of their true values which we have. The concentrations of the reactants were not measured directly. Rather the mixture was analyzed by a "densitometer" whose response to the concentrations of the reactants is

$$y = 1 + \theta_9 s_1 + \theta_{10} s_2$$

where $\theta_9 = 1 \pm 0.05$ and $\theta_{10} = 2 \pm 0.05$. The differences between the predicted and observed responses of the densitometer are assumed to be normally distributed so that least squares is used to fit the model. Bard employs an "explicit" method for integrating these differential equations, that is, the equations are approximated by a finite difference scheme like

$$\begin{array}{rcl} s_1(t_{n+1}) & = & s_1(t_n) \\ & -h & \theta_1 \exp(-\theta_2/T)(s_1(t_n) - e^{-1000/T} s_2(t_n)^2)/(1 + \theta_3 \exp(-\theta_4/T) s_1(t_n))^2 \\ s_2(t_{n+1}) & = & s_2(t_n) \\ + & 2h & \theta_1 \exp(-\theta_2/T)(s_1(t_n) - e^{-1000/T} s_2(t_n)^2)/(1 + \theta_3 \exp(-\theta_4/T) s_1(t_n))^2 \\ & (1.3) & \end{array}$$

over the time period t_n to t_{n+1} of length h. Equations 2 are called explicit because the values of s_1 and s_2 at time t_{n+1} are given explicitly in terms of the values of s_1 and s_2 at time t_n.

The advantage of using an explicit scheme for integrating the model differential equations is that the derivatives of the model functions with respect to the model parameters also satisfy differential equations – called sensitivity equations (Bard pg 227-229). It is possible to integrate these equations as well as the model equations to get values for the derivatives. However this involves generating a lot of extra code as well as carrying out a lot of extra calculations. Since with AD Model Builder it is not necessary to produce any code for derivative calculations it is possible to employ alternate schemes for integrating the differential equations.

Let $A = \theta_1 \exp(-\theta_2/T)$, $B = \exp(-1000/T)$, and $C = (1 + \theta_3 \exp(-\theta_4/T) s_1)^2$ In terms of A and C we can replace explicit finite difference scheme by the semi-implicit scheme

$$s_1(t_{n+1}) \quad = \quad s_1(t_n) - hA(s_1(t_{n+1}) - Bs_2^2(t_{n+1}))/C$$

$$s_2(t_{n+1}) = s_2(t_n) + 2hA(s_1(t_{n+1}) - Bs_2(t_n)s_2(t_{n+1}))/C \tag{1.4}$$

Now let $D = hA/C$ and solve equations (3) for $s_1(t_{n+1})$ and $s_2(t_{n+1})$ to obtain

$$\begin{aligned} s_1(t_{n+1}) &= (s_1(t_n) + DBs_2(t_n))/(1 + D) \\ s_2(t_{n+1}) &= (s_2(t_n) + 2Ds_1(t_n))/(1 + (2DBs_2(t_n))) \end{aligned} \tag{1.5}$$

Implicit and semi-implicit schemes tend to be more stable than explicit schemes over large time steps and large values of some of the model parameters. This stability is especially important when fitting nonlinear models because the algorithms for function minimization will pick very large or "bad" values of the parameters from time to time and the minimization procedure will generally perform better when a more stable scheme is employed.

```
DATA_SECTION
  init_matrix Data(1,10,1,3)
  init_vector T(1,3) // the initial temperatures for the three runs
  init_vector stepsize(1,3)  // the stepsize to use for numerical integration
  matrix data(1,3,1,10)
  matrix sample_times(1,3,1,10) // times at which reaction was sampled
  vector x0(1,3)           // the beginning time for each of the three
                           // runs
  vector x1(1,3)           // the ending time for each of the three runs
  // for each of the three runs

PARAMETER_SECTION
  init_vector theta(1,10)   // the model parameters
  matrix init_conc(1,3,1,2) // the initial concentrations of the two
                            // reactants over three time periods
  vector instrument(1,2)    // determines the response of the densitometer
  matrix y_samples(1,10,1,2)// the predicted concentrations of the two
                            // reactants at the ten sampling periods
                            // obtained by integrating the differential
                            // equations
  vector diff(1,10)         // the difference between the observed and
                            // readings of the densitometer
  objective_function_value f  // the log_likelihood function
  number bayes_part         // the Bayesian contribution
  number y2
  number x_n
  vector y_n(1,2)
  vector y_n1(1,2)
  number A  // A B C D hold some common subexpressions
  number B
  number C
  number D
PRELIMINARY_CALCS_SECTION
  data=trans(Data);   // it is more convenient to work with the transformed
                 // matrix
PROCEDURE_SECTION

    // set up the begining and ending times for the three runs
    x0(1)=0;
    x1(1)=90;
```

```
x0(2)=0;
x1(2)=18;
x0(3)=0;
x1(3)=4.5;
// set up the sample times for each of the three runs
sample_times(1).fill_seqadd(0,10);  // fill with 0,10,20,...,90
sample_times(2).fill_seqadd(0,2);   // fill with 0,2,4,...,18
sample_times(3).fill_seqadd(0,0.5); // fill with 0,0.5,1.0,...,4.5

// set up the initial concentrations of the two reactants for
// each of the three runs
init_conc(1,1)=theta(5);
init_conc(1,2)=theta(6);
init_conc(2,1)=theta(7);
init_conc(2,2)=0.0;     // the initial concentrations is known to be 0
init_conc(3,1)=0.0;     // the initial concentrations is known to be 0
init_conc(3,2)=theta(8);

// coefficients which determine the response of the densitometer
instrument(1)=theta(9);
instrument(2)=theta(10);
f=0.0;
for (int run=1;run<=3;run++)
{
   // integrate the differential equations to get the predicted
   // values for the y_samples
  int nstep=(x1(run)-x0(run))/stepsize(run);
  nstep++;
  double h=(x1(run)-x0(run))/nstep; // h is the stepsize for integration

  int is=1;
  // get the initial conditions for this run
  x_n=x0(run);
  y_n=init_conc(run);
  for (int i=1;i<=nstep+1;i++)
  {
    // gather common subexpressions
    y2=y_n(2)*y_n(2);
    A=theta(1)*exp(-theta(2)/T(run));
    B=exp(-1000/T(run));
    C=(1+theta(3)*exp(-theta(4)/T(run))*y_n(1));
    C=C*C;
    D=h*A/C;
    // get the y vector for the next time step
    y_n1(1)=(y_n(1)+D*B*y2)/(1.+D);
    y_n1(2)=(y_n(2)+2.*D*y_n(1))/(1.+(2*D*B*y_n(2)));

    // if an observation occurred during this time period save
    // the predicted value
    if (is <=10)
    {
      if (x_n<=sample_times(run,is) && x_n+h >= sample_times(run,is))
      {
        y_samples(is++)=y_n;
      }
    }
    x_n+=h;  // increment the time step
    y_n=y_n1;  // update the y vector for the next step
```

```
  }
  diff=(1.0+y_samples*instrument)-data(run); //differences between the
             // predicted and observed values of the densitometer
  f+=diff*diff;  // sum of squared differences
}
// take the log of f and multiply by nobs/2 to get log-likelihood
f=15.*log(f); // This is (number of obs)/2. It is wrong in Bard (pg 236).

// Add the Bayesian stuff
bayes_part=0.0;
for  (int i=5;i<=9;i++)
{
  bayes_part+=(theta(i)-1)*(theta(i)-1);
}
bayes_part+=(theta(10)-2)*(theta(10)-2);
f+=1./(2.*.05*.05)*bayes_part;
```

AD Model Builder produces a report containing values, standard deviations, and correlation matrix of the parameter estimates. As discussed below any parameter or group of parameters can easily be included in this report. For models with a large number of parameters this report can be a bit unwieldly so options are provided to exclude parameters from the report if desired.

index		value	std dev	1	2	3	4	5	6	7	8	9	10
1	theta	1.37e+00	2.09e-01	1									
2	theta	1.12e+03	7.70e+01	0.95	1								
3	theta	1.80e+00	7.95e-01	0.9	0.98	1							
4	theta	3.58e+02	1.94e+02	0.91	0.98	0.99	1						
5	theta	1.00e+00	4.49e-02	0.20	0.28	0.12	0.17	1					
6	theta	9.94e-01	2.99e-02	-0.42	-0.35	-0.25	-0.22	-0.58	1				
7	theta	9.86e-01	2.59e-02	0.01	0.22	0.22	0.28	0.26	0.42	1			
8	theta	1.02e+00	1.69e-02	-0.38	-0.34	-0.36	-0.30	0.09	0.63	0.34	1		
9	theta	1.00e+00	2.59e-02	-0.02	-0.23	-0.23	-0.30	-0.28	-0.43	-0.98	-0.37	1	
10	theta	1.97e+00	3.23e-02	0.44	0.37	0.40	0.32	-0.09	-0.65	-0.37	-0.93	0.40	1

Time series models are often used in financial modeling. For these models the parameters are often extremely badly determined. With the stable numerical environment produced by AD Model Builder it is a simple matter to fit such models.

Consider a time series of returns r_t where $t = 0, \ldots, T$, which are available from some type of financial instrument. The model assumptions are

$$r_t = \mu + \epsilon_t \quad h_t = a_0 + a_1\epsilon_{t-1}^2 + a_2 h_{t-1} \quad \text{for} \ \ 1 \le t \le T, \ \ a_0 \ge 0, \ \ a_1 \ge 0, \ \ a_2 \ge 0$$

where the ϵ_t are independent normally distributed random variables with mean 0 and variance h_t. We assume $\epsilon_0 = 0$ and $h_0 = \sum_{i=0}^{T}(r_i - \bar{r})^2/(T+1)$ There are four initial parameters to be estimated for this model, μ, a_0, a_1, and a_2. The log-likelihood function for the vector r_t is equal to a constant plus

$$-.5 \sum_{t=1}^{T} (\ \log(h_t) + (r_t - \mu)^2/h_t\ )$$

```
DATA_SECTION
  init_int T
  init_vector r(0,T)
  vector sub_r(1,T)
  number h0
INITIALIZATION_SECTION
  a0 .1
  a1 .1
  a2 .1
PARAMETER_SECTION
  init_bounded_number a0(0.0,1.0)
  init_bounded_number a1(0.0,1.0,2)
  init_bounded_number a2(0.0,1.0,3)
  init_number Mean
  vector eps2(1,T)
  vector h(1,T)
  objective_function_value log_likelihood
PRELIMINARY_CALCS_SECTION
  h0=square(std_dev(r));   // square forms the element-wise square
  sub_r=r(1,T);    // form a subvector so we can use vector operations
  Mean=mean(r);    // calculate the mean of the vector r
PROCEDURE_SECTION
  eps2=square(sub_r-Mean);
  h(1)=a0+a2*h0;
  for (int t=2;t<=T;t++)
  {
    h(t)=a0+a1*eps2(t-1)+a2*h(t-1);
  }
  // calculate minus the log-likelihood function
  log_likelihood=.5*sum(log(h)+elem_div(eps2,h));  // elem_div performs
                                    // element-wise division of vectors
RUNTIME_SECTION
  convergence_criteria .1, .1, .001
  maximum_function_evaluations 20, 20, 1000
```

We have used vector operations such as `elem_div` and `sum` to simplify the code. Of course the code could also have employed loops and element-wise operations. The parameter values and standard deviation report for this model appears below.

```
 index         value      std dev       1       2       3       4
    1   a0    1.6034e-04 2.3652e-05  1.0000
    2   a1    9.3980e-02 2.0287e-02  0.1415  1.0000
    3   a2    3.7263e-01 8.2333e-02 -0.9640 -0.3309  1.0000
    4   Mean -1.7807e-04 3.0308e-04  0.0216 -0.1626  0.0144  1.0000
```

This example employs bounded initial parameters. Often it is necessary to put bounds on parameters in nonlinear modeling to ensure that the minimization is stable. In this example `a0` is constrained to lie between 0.0 and 1.0

```
init_bounded_number a0(0.0,1.0)
init_bounded_number a1(0.0,1.0,2)
init_bounded_number a2(0.0,1.0,3)
```

For linear models one can simply estimate all the model parameters simultaneously. For nonlinear models often this simple approach does not work very well. It may be necessary to keep some of the parameters fixed during the initial part of the minimization process and carry out the minimization over a subset of the parameters. The other parameters are included into the minimization process in a number of phases until all of the parameters have been included. AD Model Builder provides support for this multi-phase approach. In the declaration of any initial parameter the last number, if present, determines the phase of the minimization during which this parameter is included (becomes active). If no number is present the initial parameter becomes active in phase 1. In this case `a0` has no phase number and so becomes active in phase 1. `a1` becomes active in phase 2, and `a2` becomes active in phase 3. In this example phase 3 is the last phase of the optimization.

It is often convenient to modify aspects of the code depending on which phase of the minimization procedure is the current phase or on whether a particular initial parameter is active. The function

```
current_phase()
```

returns an integer (object of type `int`) which is the value of the current phase. The function

```
last_phase()
```

returns the value "true" ($\neq 0$) if the current phase is the last phase and false ($= 0$) otherwise. If `xxx` is the name of any initial parameter the function

```
active(xxx)
```

returns the value "true" if `xxx` is active during the current phase and false otherwise.

After the minimization of the objective function has been completed AD Model Builder calculates the estimated covariance matrix for the initial parameters as well as any other desired parameters which have been declared to be of `sd_report` type. Often these additional parameters may involve considerable additional computational overhead. If the values of these parameters are not used in calculations proper, it is possible to only calculate them during the standard deviations report phase.

```
sd_phase()
```

The `sd_phase` function returns the value "true" if we are in the standard deviations report phase and "false" otherwise. It can be used in a conditional statement to determine whether to perform calculations associated with some `sd_report` object. When estimating the parameters of a model by a multi-phase minimization procedure the default behavior of AD Model Builder is to carry out the default number of function evaluations until convergence is achieved in each stage. If we are only interested in the parameter estimates obtained after the last stage of the minimization it is often not necessary to carry out the full minimization in each stage. Sometimes considerable time can be saved by relaxing the convergence criterion in the initial stages of the optimization. The `RUNTIME_SECTION` allows the user to modify the default behavior of the function minimizer during the phases of the estimation process.

```
RUNTIME_SECTION
  convergence_criteria .1, .1, .001
  maximum_function_evaluations 20, 20, 1000
```

The `convergence_criteria` affects the criterion used by the function minimizer to decide when the optimization process has occurred. The function minimizer compares the maximum value of the vector of derivatives of the objective function with respect to the independent variables to the numbers after the `convergence_criteria` keyword. The first number is used in the first phase of the optimization, the second number in the second phase and so on. If there are more phases to the optimization than there are numbers the last number is used for the rest of the phases of the optimization. The numbers must be separated by commas. The spaces are optional. The `maximum_function_evaluations` keyword controls the maximum number of evaluations of the objective function which will be performed by the function minimizer in each stage of the minimization procedure.

It is typical of many models in natural resource management that the model tends to be rather unstable numerically and in addition some of the model parameters are often poorly determined. Notwithstanding these difficulties it is often necessary to make decisions about resource management based on the analysis provided by these models. The example provides a good opportunity for presenting some more advanced features of AD Model Builder which are designed to overcome these difficulties.

The Schaeffer – Pella-Tomlinson model is employed in fisheries management. The model assumes that the total biomass of an exploited fish stock satisfies an ordinary differential equation of the form

$$\frac{dB}{dt} = rB\Big(1 - \big(\frac{B}{k}\big)^{m-1}\Big) - FB \qquad \text{where} \quad m > 1 \tag{1.6}$$

(Hilborn and Walters page 303) where B is the biomass, F is the instantaneous fishing mortality rate, r is a parameter often referred to an the intrinsic rate of increase, k is the unfished equilibrium stock size,

$$C = FB \tag{1.7}$$

is the catch rate, and m is a parameter which determines where the maximum productivity of the stock occurs. If the value of m is fixed at 2 the model is referred to as the Schaeffer model.

The explicit form of the difference equation corresponding to this differential equation is

$$B_{t+\delta} = B_t + rB_t\delta - rB_t\Big(\frac{B_t}{k}\Big)^{m-1}\delta - F_tB_t\delta \tag{1.8}$$

To get a semi-implicit form of this difference equation which has better numerical stability than the explicit version we replace some of the terms B_t on the right hand side of 1.8 by $B_{t+\delta}$ to get

$$B_{t+\delta} = B_t + rB_t\delta - rB_{t+\delta}\Big(\frac{B_t}{k}\Big)^{m-1}\delta - F_tB_{t+\delta}\delta \tag{1.9}$$

and solve for $B_{t+\delta}$ to give

$$B_{t+\delta} = \frac{B_t(1+r\delta)}{1+(\,r(B_t/k)^{m-1}+F_t\,)\delta} \tag{1.10}$$

The catch $C_{t+\delta}$ over the period $(t, t+\delta)$ is given by

$$C_{t+\delta} = F_tB_{t+\delta}\delta \tag{1.11}$$

The parameter k is referred to as the carrying capacity or the unfished equilibrium biomass level because it is the value that the biomass of the population will eventually assume if there is no fishing. For a given value of k the parameter m determines the level of maximum productivity, that is the level of biomass B_{MAX} for which the removals from fishing can be the greatest.

$$B_{\mathrm{MAX}} = \frac{k}{\sqrt[m-1]{m}}$$

For $m = 2$ maximum productivity is obtained by that level of fishing pressure which reduces the stock to 50% of the carrying capacity. For the data available in many real fisheries problems the parameter m is very poorly determined. It is common practice therefore to simply assume that $m = 2$. Similarly, it is commonly assumed that the carrying capacity k does not change over time even though changes such as habitat degradation may well lead to changes in k.

We want to construct a statistical model where the carrying capacity can be varying slowly over time if there appears to be any information in the fisheries data supporting this hypothesis. What is meant by slowly? The answer to this question will depend on the particular situation. For our purposes slowly means slowly enough so that the model has some chance of supplying a useful analysis of the situation at hand. We refer to this as the assumption of manageability. The point is that since we are going to use this model anyway to try and mange a resource we may as well assume that the model's assumptions are satisfied at least well enough so that we have some hope of success. This may seem extremely arbitrary, and it is. However it is not as arbitrary as assuming that the carrying capacity is constant.

We assume that $k_{i+1} = k_i \exp(\kappa_i)$ where the κ_i are independent normally distributed random variables with mean 0. and that $\log(m-1)$ is normally distributed with mean 0. The parameters $\log(k)$ are assumed to have the structure of a random walk which is the simplest type of time series. This Bayesian approach is a very simple method for including time series structure into the parameters of a nonlinear model.

We don't know the true catches C_i in each year. What we have are estimates C_i^{obs} of the catch. We assume that the quantities $\log(C_i^{obs}/C_i)$ are normally distributed with mean 0.

Finally we must deal with the fishing mortality F. Estimates of F are not obtained directly. Instead what is observed is an index of fishing mortality, in this case fishing effort. We assume that for each year we have an estimate E_i of fishing effort and that the fishing mortality rate F_i in year i Satisfies the relationship $F_i = qE_i \exp(\eta_i)$ where q is a parameter referred to as the catchability and the η_i are normally distributed random variables with mean 0.

We assume that the variance of the η_i is 10 times the variance in the observed catch errors and that the variance of the κ_i is 0.1 times the variance in the observed catch errors. We assume that the variance in $\log(m-1)$ is 0.25. Then given the data, the Bayesian posterior distribution for the model parameters is proportional to

$$-(3n-1)/2 \log\Big(\sum_{i=1}^{n} \big(\log(C_i^{obs}) - \log(C_i) \big)^2 + .1 \sum_{i=1}^{n} \eta_i^2 + 10 \sum_{i=2}^{n} \kappa_i^2 \Big) - 2. \log(m-1)^2 \tag{1.12}$$

The number of initial parameters in the model (that is the number of independent variables in the function to be minimized) is $2n+4$. For the halibut data there are 56 years of data which gives 116 parameters. As estimates of the model parameters we use the mode of the posterior distribution which can by found by minimizing -1 times expression (0.8). The covariance matrix of the model parameters are estimated by computing the inverse of the hessian of expression (0.8) at the minimum. The template for the model follows. To improve the readability the entire template has been included. The various sections are discussed below.

```
DATA_SECTION
  init_int nobs;
  init_matrix data(1,nobs,1,3)
  vector obs_catch(1,nobs);
  vector cpue(1,nobs);
  vector effort(1,nobs);
  number avg_effort
INITIALIZATION_SECTION
  m 2.
  beta 1.
  r 1.
PARAMETER_SECTION
  init_bounded_number q(0.,1.)
```

```
  init_bounded_number beta(0.,5.)
  init_bounded_number r(0.,5,2)
  init_number log_binit(2)
  init_bounded_dev_vector effort_devs(1,nobs,-5.,5.,3)
  init_bounded_number m(1,10.,4)
  init_bounded_vector k_devs(2,nobs,-5.,5.,4)
  number binit
  vector pred_catch(1,nobs)
  vector biomass(1,nobs)
  vector f(1,nobs)
  vector k(1,nobs)
  vector k_trend(1,nobs)
  sdreport_number k_1
  sdreport_number k_last
  sdreport_number k_change
  sdreport_number k_ratio
  sdreport_number B_projected
  number tmp_mort;
  number bio_tmp;
  number c_tmp;
  objective_function_value ff;
PRELIMINARY_CALCS_SECTION
  // get the data out of the data matrix into
  obs_catch=column(data,2);
  cpue=column(data,3);
  // divide the catch by the cpue to get the effort
  effort=elem_div(obs_catch,cpue);
  // normalize the effort and save the average
  double avg_effort=mean(effort);
  effort/=avg_effort;
  cout << " beta" << beta << endl;
PROCEDURE_SECTION
  // calculate the fishing mortality
  calculate_fishing_mortality();
  // calculate the biomass and predicted catch
  calculate_biomass_and_predicted_catch();
  // calculate the objective function
  calculate_the_objective_function();

FUNCTION calculate_fishing_mortality
  // calculate the fishing mortality
  f=q*effort;
  if (active(effort_devs)) f=elem_prod(f,exp(effort_devs));

FUNCTION calculate_biomass_and_predicted_catch
  // calculate the biomass and predicted catch
  if (!active(log_binit))
  {
    log_binit=log(obs_catch(1)/(q*effort(1)));
  }
  binit=exp(log_binit);
  biomass[1]=binit;
  if (active(k_devs))
  {
    k(1)=binit/beta;
    for (int i=2;i<=nobs;i++)
    {
      k(i)=k(i-1)*exp(k_devs(i));
```

```
    }
  }
  else
  {
    // set the whole vector equal to the constant k value
    k=binit/beta;
  }
  // only calculate these for the standard deviation report
  if (sd_phase)
  {
    k_1=k(1);
    k_last=k(nobs);
    k_ratio=k(nobs)/k(1);
    k_change=k(nobs)-k(1);
  }
  // two time steps per year desired
  int nsteps=2;
  double delta=1./nsteps;
  // Integrate the logistic dynamics over n time steps per year
  for (int i=1; i<=nobs; i++)
  {
    bio_tmp=1.e-20+biomass[i];
    c_tmp=0.;
    for (int j=1; j<=nsteps; j++)
    {
      //This is the new biomass after time step delta
      bio_tmp=bio_tmp*(1.+r*delta)/
        (1.+ (r*pow(bio_tmp/k(i),m-1.)+f(i))*delta );
      // This is the catch over time step delta
      c_tmp+=f(i)*delta*bio_tmp;
    }
    pred_catch[i]=c_tmp;        // This is the catch for this year
    if (i<nobs)
    {
      biomass[i+1]=bio_tmp;// This is the biomass at the begining of the next
    }                      // year
    else
    {
      B_projected=bio_tmp; // get the projected biomass for std dev report
    }
  }

FUNCTION calculate_the_objective_function
  if (!active(effort_devs))
  {
    ff=nobs/2.*log(norm2(log(obs_catch)-log(1.e-10+pred_catch)));
  }
  else if(!active(k_devs))
  {
    ff= .5*(size_count(obs_catch)+size_count(effort_devs))*
      log(norm2(log(obs_catch)-log(1.e-10+pred_catch))
      +0.1*norm2(effort_devs));
  }
  else
  {
    ff= .5*( size_count(obs_catch)+size_count(effort_devs)
      +size_count(k_devs) )*
      log(norm2(log(obs_catch)-log(pred_catch))
```

```
    + 0.1*norm2(effort_devs)+10.*norm2(k_devs));
}
// Bayesian contribution for Pella Tomlinson m
ff+=2.*square(log(m-1.));
if (current_phase()<3)
{
  ff+=1000.*square(log(mean(f)/.4));
}
```

The data are contained in three columns, with the catch and catch per unit effort data contained in the second and third columns. The matrix `data` is defined in order to read the data. The second and third columns of `data` which we are interested in will then be put into the vectors `obs_catch` and `cpue`. (Later we get the fishing effort by dividing the `obs_catch` by the `cpue`.)

```
DATA_SECTION
  init_int nobs
  init_matrix data(1,nobs,1,3)
  vector obs_catch(1,nobs)
  vector cpue(1,nobs)
  vector effort(1,nobs)
  number avg_effort
```

The `INITIALIZATION_SECTION` is used to define default values for some model parameters if the standard default provided by AD Model Builder is not acceptable. If the model finds the parameter file (whose default name is `admodel.par`) it will read in the initial values for the parameters from there. Otherwise the default values will be used unless the parameters appear in the `INITIALIZATION_SECTION` in which case those values will be used.

```
INITIALIZATION_SECTION
  m 2.
  beta 1.
  r 1.
```

The `PARAMETER_SECTION` for this model introduces several new features of AD Model Builder. The statement `init_bounded_number r(0.,5.,2)` declares an initial parameter whose value will be constrained to lie between 0.0 and 5.0. It is often necessary to put bounds on the initial parameters in nonlinear models to get stable model performance. This is accomplished in AD Model Builder simply by declaring the initial parameter to be bounded and providing the desired bounds. The default initial value for a bounded object is the average of the lower and upper bounds.

The third number 2 in the declaration determines that this initial parameter will not be made active until the second phase of the minimization. This introduces the concept of phases in the minimization process.

As soon as nonlinear statistical models become a bit complicated one often finds that simply attempting to estimate all the parameters simultaneously does not work very well. In short “you can’t get there from here”. A better strategy is to keep some of the parameters fixed and to first minimize the function with respect to the other parameters. More parameters are added in a stepwise relaxation process. In AD Model Builder each step of this relaxation process is termed a phase. The parameter `r` is not allowed to vary until the second phase. Initial parameters which are allowed to vary will be termed active. In the first phase the active parameter are `beta` and `q`. The default phase for an initial parameter is phase 1 if no phase number is included in its declaration. The phase number for an initial parameter is the last number in the declaration for that parameter. The general order for the arguments int the definition of any initial parameter is the size data for a vector or matrix object if needed, the bounds for a bounded object if needed, followed by the phase number if desired.

It is often a difficult problem to decide what the order of relaxation for the initial parameters should be. This must sometimes be done by trial and error. HoweverAD Model Builder makes the process a lot simpler. One only needs to change the phase numbers of the initial parameters int the `PARAMETER_SECTION` and recompile the program.

Often in statistical modeling it is useful to regard a vector of quantities x_i as consisting of an overall mean, μ, and a set of deviations from that mean, δ_i, so that

$$x_i = \mu + \delta_i \qquad \text{where} \quad \sum_i \delta_i = 0$$

AD Model Builder provides support for this modeling construction with the `init_bounded_dev_vector` declaration. The components of an object created by this declaration will automatically sum to 0 without any user intervention. The line

```
init_bounded_dev_vector effort_devs(1,nobs,-5.,5.,3)
```

declares effort_devs to be this kind of object. The bounds `-5.,5.` control the range of the deviations. Putting reasonable bounds on such deviations often improves the stability of the estimation procedure.

AD Model Builder has `sdreport_number`, `sdreport_vector`, and `sdreport_matrix` declarations in the `PARAMETER_SECTION`. These objects behave the same as `number`, `vector`, and `matrix` objects with the additional property that they are included in the report of the estimated standard deviations and correlation matrix.

For example merely by including the statement `sdreport_number B_projected` one can obtain the estimated standard deviation of the biomass projection for the next year. (Of course you must also set `B_projected` equal to the projected biomass. This is done in the `PROCEDURE_SECTION`.)

```
PARAMETER_SECTION
  init_bounded_number q(0.,1.)
  init_bounded_number beta(0.,5.)
  init_bounded_number r(0.,5,2)
  init_number log_binit(2)
  init_bounded_dev_vector effort_devs(1,nobs,-5.,5.,3)
  init_bounded_number m(1,10.,4)
  init_bounded_vector k_devs(2,nobs,-5.,5.,4)
  number binit
```

```
vector pred_catch(1,nobs)
vector biomass(1,nobs)
vector f(1,nobs)
vector k(1,nobs)
vector k_trend(1,nobs)
sdreport_number k_1
sdreport_number k_last
sdreport_number k_change
sdreport_number k_ratio
sdreport_number B_projected
number tmp_mort;
number bio_tmp;
number c_tmp;
objective_function_value ff;
```

The `PRELIMINARY_CALCS_SECTION` carries out a few simple operations on the data. The model expects to have catch and effort data, but the input file contained catch and cpue (catch/effort) data. We divide the catch data by the cpue data to get the effort data. The AUTODIF operation `elem_div` which performs element-wise divisions of vector objects is used. As usual the same thing could have been accomplished by employing a loop and writing element-wise code. The effort data are then normalized, that is, they are divided by their average so that their average becomes 1. This is done so that we have a good idea what the catchability parameter `q` should be to give reasonable values for the fishing mortality rate (since $F = qE$).

Notice that the `PRELIMINARY_CALCS_SECTION` section is C++ code so that statements must be ended with a `;`. extract a column from a matrix

```
PRELIMINARY_CALCS_SECTION
  // get the data out of the data matrix into
  obs_catch=column(data,2);
  cpue=column(data,3);
  // divide the catch by the cpue to get the effort
  effort=elem_div(obs_catch,cpue);
  // normalize the effort and save the average
  double avg_effort=mean(effort);
  effort/=avg_effort;
```

The `PROCEDURE_SECTION` contains several new AD Model Builder features. Some have to do with the notion of carrying out the minimization in a number of steps or phases. The line

```
if (active(effort_devs)) f=elem_prod(f,exp(effort_devs));
```

introduces the `active` function. This function can be used on any initial parameter and will return a value "true" if that parameter is active in the current phase. The idea here is that if the initial parameters `effort_devs` are not active then since their value is 0 carrying out the calculations will have no effect and we can save time by avoiding the calculations. The `active` function is also used in the statement

```
if (!active(log_binit))
{
  log_binit=log(obs_catch(1)/(q*effort(1)));
```

```
}
```

The idea is that if the log_binit initial parameter (this is the logarithm of the biomass at the beginning of the first year) is not active then we set it equal to the value which produces the observed catch (using the relationship $C = qEB$ so that $B = C/(qE)$. The active function is also used in the calculations of the objective function so that unnecessary calculations are avoided.

The following code helps to deal with convergence problems in this type of nonlinear model. The problem is that the starting parameter values are often so bad that the optimization procedure will try to make the population very large and the exploitation rate very small because this is the best local solution near the starting parameter values. To circumvent this problem we include a penalty function to keep the average value of the fishing mortality rate f close to 0.2 during the first two phases of the minimization. In the final phase the size of the penalty term is reduced to a very small value. The function current_phase() returns the value of the current phase of the minimization.

```
if (current_phase()<3)
 {
   ff+=1000.*square(log(mean(f)/.4));
 }
```

Subroutines or functions are used to improve the organization of the code. The code for the main part of the PROCEDURE_SECTION which invokes the FUNCTIONS should be placed at the top of the PROCEDURE_SECTION.

```
PROCEDURE_SECTION
  // calculate the fishing mortality
  calculate_fishing_mortality();
  // calculate the biomass and predicted catch
  calculate_biomass_and_predicted_catch();
  // calculate the objective function
  calculate_the_objective_function();
```

There are three user-defined functions called at the beginning of the PROCEDURE_SECTION The code to define the FUNCTIONS comes next. To define a function whose name is name the template directive FUNCTION name is used. Notice that no parentheses () are used in the definition of the function, but to call the function the statement takes the form name();

This section describes a simple catch-at age model. The data input to this model include estimates of the numbers at age caught by the fishery each year and estimates the fishing effort each year. This example introduces AD Model Builder's ability to automatically calculate profile likelihoods for carrying out Bayesian inference. To cause the profile likelihood calculations to be carried out use the -lprof command line argument.

Let i index fishing years $1 \le i \le n$ and j index age classes with $1 \le j \le r$. The instantaneous fishing mortality rate is assumed to have the form $F_{ij} = qE_i s_j \exp(\delta_i)$ where q is called the catchability, E_i is the observed fishing effort, s_j is an age-dependent effect termed the selectivity, and the δ_i are deviations from the expected relationship between the observed fishing effort and the resulting fishing mortality. The δ_i are assumed to be normally distributed with mean 0. The instantaneous natural mortality rate M is assumed to be independent of year and age class. It is not estimated in this version of the model. The instantaneous total mortality rate is given by $Z_{ij} = F_{ij} + M$. The survival rate is given by $S_{ij} = \exp(-Z_{ij})$. The number of age class j fish in the population in year i is denoted by N_{ij}. The relationship $N_{i+1,j+1} = N_{ij}S_{ij}$ is assumed to hold. Note that using this relationship if one knows S_{ij} then all the N_{ij} can be calculated from knowledge of the initial population in year 1, $N_{11}, N_{12}, \ldots, N_{1r}$ and knowledge of the recruitment in each year $N_{21}, N_{31}, \ldots N_{n1}$.

The purpose of the model is to estimate quantities of interest to managers such as the population size and exploitation rates and to make projections about the population. In particular we can get an estimate of the numbers of fish in the population in year $n+1$ for age classes 2 or greater from the relationship $N_{n+1,j+1} = N_{nj}S_{nj}$. If we have estimates m_j for the mean weight at age j, then the projected biomass level B_{n+1} of age class 2+ fish for year $n+1$ can be computed from the relationship $B_{n+1} = \sum_{j=2}^{r} m_j N_{n+1,j}$.

Besides getting a point estimate for quantities of interest like B_{n+1} we also want to get an idea of how well determined the estimate is. AD Model Builder has completely automated the process of deriving good confidence limits for these parameters in a Bayesian context. One simply needs to declare the parameter to be of type `likeprof_number`. The results are given in the section on Bayesian inference.

The code for the catch-at-age model is:

```
DATA_SECTION
  // the number of years of data
  init_int nyrs
  // the number of age classes in the population
  init_int nages
  // the catch-at-age data
  init_matrix obs_catch_at_age(1,nyrs,1,nages)
  //estimates of fishing effort
  init_vector effort(1,nyrs)
  // natural mortality rate
  init_number M
  // need to have relative weight at age to calculate biomass of 2+
  vector relwt(2,nages)
INITIALIZATION_SECTION
  log_q -1
  log_P 5
PARAMETER_SECTION
  init_number log_q(1)   // log of the catchability
  init_number log_P(1)  // overall population scaling parameter
  init_bounded_dev_vector log_sel_coff(1,nages-1,-15.,15.,2)
  init_bounded_dev_vector log_relpop(1,nyrs+nages-1,-15.,15.,2)
  init_bounded_dev_vector effort_devs(1,nyrs,-5.,5.,3)
  vector log_sel(1,nages)
```

```
  vector log_initpop(1,nyrs+nages-1);
  matrix F(1,nyrs,1,nages)   // the instantaneous fishing mortality
  matrix Z(1,nyrs,1,nages)   // the instantaneous total mortality
  matrix S(1,nyrs,1,nages)   // the survival rate
  matrix N(1,nyrs,1,nages)   // the predicted numbers at age
  matrix C(1,nyrs,1,nages)   // the predicted catch at age
  objective_function_value f
  sdreport_number avg_F
  sdreport_vector predicted_N(2,nages)
  sdreport_vector ratio_N(2,nages)
  likeprof_number pred_B
PRELIMINARY_CALCS_SECTION
  // this is just to invent some relative average
  // weight numbers
  relwt.fill_seqadd(1.,1.);
  relwt=pow(relwt,.5);
  relwt/=max(relwt);
PROCEDURE_SECTION
  // example of using FUNCTION to structure the procedure section
  get_mortality_and_survival_rates();

  get_numbers_at_age();

  get_catch_at_age();

  evaluate_the_objective_function();

FUNCTION get_mortality_and_survival_rates
  // calculate the selectivity from the sel_coffs
  for (int j=1;j<nages;j++)
  {
    log_sel(j)=log_sel_coff(j);
  }
  // the selectivity is the same for the last two age classes
  log_sel(nages)=log_sel_coff(nages-1);

  // This is the same as F(i,j)=exp(log_q)*effort(i)*exp(log_sel(j));
  F=outer_prod(mfexp(log_q)*effort,mfexp(log_sel));
  if (active(effort_devs))
  {
    for (int i=1;i<=nyrs;i++)
    {
      F(i)=F(i)*exp(effort_devs(i));
    }
  }
  // get the total mortality
  Z=F+M;
  // get the survival rate
  S=mfexp(-1.0*Z);

FUNCTION get_numbers_at_age
  log_initpop=log_relpop+log_P;
  for (int i=1;i<=nyrs;i++)
  {
    N(i,1)=mfexp(log_initpop(i));
  }
  for (int j=2;j<=nages;j++)
  {
```

```
    N(1,j)=mfexp(log_initpop(nyrs+j-1));
  }
  for (i=1;i<nyrs;i++)
  {
    for (j=1;j<nages;j++)
    {
      N(i+1,j+1)=N(i,j)*S(i,j);
    }
  }
  // calculated predicted numbers at age for next year
  for (j=1;j<nages;j++)
  {
    predicted_N(j+1)=N(nyrs,j)*S(nyrs,j);
    ratio_N(j+1)=predicted_N(j+1)/N(1,j+1);
  }
  // calculate predicted biomass for profile
  // likelihood report
  pred_B=predicted_N *relwt;

FUNCTION get_catch_at_age
  C=elem_prod(elem_div(F,Z),elem_prod(1.-S,N));

FUNCTION evaluate_the_objective_function
  // penalty functions to ‘‘regularize ’’ the solution
  f+=.01*norm2(log_relpop);
  avg_F=sum(F)/double(size_count(F));
  if (last_phase())
  {
    // a very small penalty on the average fishing mortality
    f+= .001*square(log(avg_F/.2));
  }
  else
  {
    // use a large penalty during the initial phases to keep the
    // fishing mortality high
    f+= 1000.*square(log(avg_F/.2));
  }
  // errors in variables type objective function with errors in
  // the catch at age and errors in the effort fishing mortality
  // relationship
  if (active(effort_devs)
  {
    // only include the effort_devs in the objective function if
    // they are active parameters
    f+=0.5*double(size_count(C)+size_count(effort_devs))
      * log( sum(elem_div(square(C-obs_catch_at_age),.01+C))
      + 0.1*norm2(effort_devs));
  }
  else
  {
    // objective function without the effort_devs
    f+=0.5*double(size_count(C))
      * log( sum(elem_div(square(C-obs_catch_at_age),.01+C)));
  }
REPORT_SECTION
  report << "Estimated numbers of fish " << endl;
  report << N << endl;
  report << "Estimated numbers in catch " << endl;
```

```
report << C << endl;
report << "Observed numbers in catch " << endl;
report << obs_catch_at_age << endl;
report << "Estimated fishing mortality " << endl;
report << F << endl;
```

This model employs several instances of the `init_bounded_dev_vector` type. This type consists of a vector of numbers which sum to 0, that is they are deviations from a common mean, and are bounded. For example the quantities `log_relpop` are used to parameterize the logarithm of the variations in year class strength of the fish population. Putting bounds on the magnitude of the deviations helps to improve the stability of the model. The bounds are from -15.0 to 15.0 which gives the estimates of relative year class strength a dynamic range of exp(30.0).

The `FUNCTION` keyword has been employed a number of times in the `PARAMETER_SECTION` to help structure the code. A function is defined simply by using the `FUNCTION` keyword followed by the name of the function.

```
FUNCTION get_mortality_and_survival_rates
```

Don't include the parentheses or semicolon here. To use the function simply write its name in the procedure section.

```
get_mortality_and_survival_rates();
```

You must include the parentheses and the semicolon here.

The `REPORT_SECTION` shows how to generate a report for an AD Model Builder program. The default report generating machinery utilizes the C++ stream formalism. You don't need to know much about streams to make a report, but a few comments are in order. The stream formalism associates stream object with a file. In this case the stream object associated with the AD Model Builder report file is `report`. To write an object `xxx` into the report file you insert the line

```
report << xxx;
```

into the `REPORT_SECTION`. If you want to skip to a new line after writing the object you can include the stream manipulator `endl` as in

```
report << "Estimated numbers of fish " << endl;
```

Notice that the stream operations know about common C objects such as strings, so that it is a simple matter to put comments or labels into the report file.

AD Model Builder enables one to quickly build models with large numbers of parameters – this is especially useful for employing Bayesian analysis. Traditionally however it has been difficult to interpret the results of analysis using such models. In a Bayesian context the results are represented by the posterior probability distribution for the model parameters.

To get exact results from the posterior distribution it is necessary to evaluate integrals over large dimensional spaces and this can be computationally intractable. AD Model Builder provides an approximations to these integrals in the form of the profile likelihood. The profile likelihood can be used to estimates for extreme values (such as estimating a value β so that for a parameter b the probability that $b < \beta \approx 0.10$ or the probability that $b > \beta \approx 0.10$) for any model parameter. To use this facility simply declare the parameter of interest to be of type `likeprof_number` in the `PARAMETER_SECTION` and assign the correct value to the parameter in the `PROCEDURE_SECTION`.

The code for the catch at age model estimates the profile likelihood for the projected biomass of age class 2+ fish. (Age class 2+ has been used to avoid the extra problem of dealing with the uncertainty of the recruitment of age class 1 fish). As a typical application of the method, the user of the model can estimate the probability that the biomass of fish for next year will be larger or smaller than a certain value. Estimates like these are obviously of great interest to managers of natural resources.

The profile likelihood report for a variable is in a file with the same name as the variable (truncated to eight letters, if necessary, with the suffix `.PLT` appended). For this example the report is in the file `PRED_B.PLT`. Part of the file is shown here.

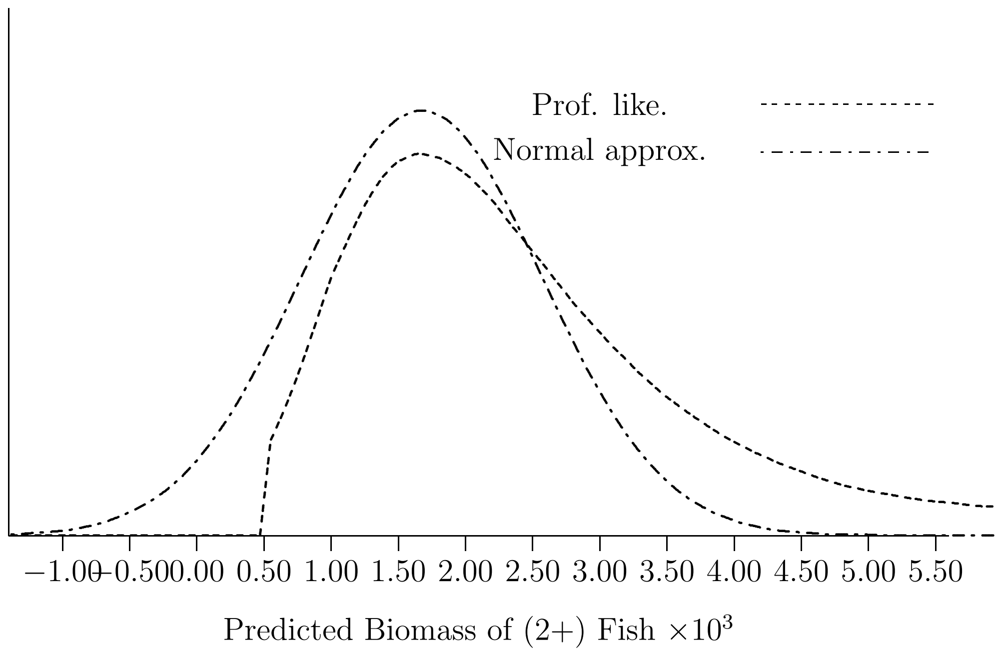
```
pred_B:
Profile likelihood
 -1411.23 1.1604e-09
 -1250.5 1.71005e-09
 -1154.06 2.22411e-09
  ..................      // skip some here
  ..................
 278.258 2.79633e-05
 324.632 5.28205e-05
 388.923 6.89413e-05
 453.214 8.84641e-05
 517.505 0.0001116
 581.796 0.000138412
  ..................
  ..................
 1289 0.000482459
 1353.29 0.000494449
 1417.58 0.000503261
 1481.87 0.000508715
 1546.16 0.0005107
 1610.45 0.000509175
 1674.74 0.000504171
 1739.03 0.000490788
 1803.32 0.000476089
 1867.61 0.000460214
 1931.91 0.000443313
 1996.2 0.000425539
 2060.49 0.000407049
 2124.78 0.000388
 2189.07 0.00036855
  ..................
  ..................
 4503.55 2.27712e-05
```

```
 4599.98 2.00312e-05
 4760.71 1.48842e-05
 4921.44 1.07058e-05
 5082.16 7.45383e-06
  ..................
  ..................
 6528.71 6.82689e-07
 6689.44 6.91085e-07
 6850.17 7.3193e-07
Minimum width confidence limits:
        significance level  lower bound  upper bound
               0.90            572.537     3153.43
               0.95            453.214     3467.07
               0.975           347.024     3667.76

One sided confidence limits for the profile likelihood:

The probability is     0.9 that pred_B is greater than 943.214
The probability is    0.95 that pred_B is greater than 750.503
The probability is   0.975 that pred_B is greater than 602.507

The probability is     0.9 that pred_B is less than 3173.97
The probability is    0.95 that pred_B is less than 3682.75
The probability is   0.975 that pred_B is less than 4199.03
```

The file contains the probability density function and the approximate confidence limits for the the profile likelihood and the normal approximation. Since the format is the same for both, we only discuss the profile likelihood here. The first part of the report contains pairs of numbers (x_i, y_i) which consist of values of the parameter in the report (in this case `PRED_B` and the estimated value for the probability density associated with that parameter at the point. The probability that the parameter lies in the interval $x_r \leq x \leq x_s$) where $x_r < x_s$ can be estimated from the sum

$$\sum_{i=r}^{s}(x_{i+1} - x_i)y_i.$$

The reports of the one and two sided confidence limits for the parameter were produced this way. Also a plot of y_i verses x_i gives the user an indication of what the probability distribution of the parameter looks like.



The the profile likelihood indicates the fact that the biomass can not be less than zero. The normal approximation is not very useful for calculating the probability that the biomass is very low – a question of great interest to managers who are probably not going to be impressed by the knowledge that there is an estimated probability of 0.975 that the biomass is greater than -52.660.

```
One sided confidence limits for the normal approximation

The probability is     0.9 that pred_B is greater than  551.235
The probability is    0.95 that pred_B is greater than  202.374
The probability is   0.975 that pred_B is greater than  -52.660
```

The functions `set_stepnumber()` and `set_stepsize()` can be used to modify the number of points used to approximate the profile likelihood or to change the stepsize between the points. This can be carried out in the `PRELIMINARY_CALCS_SECTION`. If `u` has been declared to be of type `likeprof_number`

```
PRELIMINARY_CALCS_SECTION
  u.set_stepnumber(10);   // default value is 8
  u.set_stepsize(0.2);    // default value is 0.5
```

will set the number of steps equal to 21 (from -10 to 10) and will set the step size equal to 0.2 times the estimated standard deviation for the parameter `u`.

The following code fragment illustrates how the files used for input of the data and parameter values can be changed. This code has been taken from the example `catage.tpl` and modified. In the `DATA_SECTION`, the data are first read in from the file `catch.dat`. Then the effort data are read in from the file `effort.dat`. The remainder of the data are read in from the file `catch.dat`. It is necessary to save the current file position in an object of type `streampos`. This object is used to position the file properly. The escape sequence `!!` can be used to include one line of the users's code into the `DATA_SECTION` or `PARAMETER_SECTION`. It is more compact than the `LOCAL_CALCS` construction.

```
DATA_SECTION
 // will read data from file catchdat.dat
 !! ad_comm::change_datafile_name("catchdat.dat");
  init_int nyrs
  init_int nages
  init_matrix obs_catch_at_age(1,nyrs,1,nages)
 // now read the effort data from the file effort.dat and save the current
 // file position in catchdat.dat in the object tmp
 !! streampos tmp = ad_comm::change_datafile_name("effort.dat");
  init_vector effort(1,nyrs)
 // now read the rest of the data from the file catchdat.dat
 // including the ioption argument tmp will reset the file to that position
 !! ad_comm::change_datafile_name("catchdat.dat",tmp);
  init_number M

 // ....

PARAMETER_SECTION
 // will read parameters from file catch.par
 !! ad_comm::change_parfile_name("catch.par");
```

If `v` is a vector object then for integers `l` and `u` the expression `v(l,u)` is a vector object of the same type with minimum valid index `l` and maximum valid index `u` (Of course `l` and `u` must be within the valid index range for `v` and `l` must be less than or equal to `u`. The subvector formed by this operation ican be used on both sides of the equals sign in an arithmetic expression. The number of loops which must be written can be significantly reduced in this manner. We shall use the subvector operator to remove some of the loops in the catch-at-age model code.

```
// calculate the selectivity from the sel_coffs
for (int j=1;j<nages;j++)
{
  log_sel(j)=log_sel_coff(j);
}
// the selectivity is the same for the last two age classes
log_sel(nages)=log_sel_coff(nages-1);

// same code using the subvector operation
log_sel(1,nage-1)=log_sel_coff;
// the selectivity is the same for the last two age classes
log_sel(nages)=log_sel_coff(nages-1);
```

Notice that log_sel(1,nage-1) is not a distinct vector from log_sel. This means that an assignment to log_sel(1,nage-1) is an assignment to a part of log_sel. The next example is a bit more complicated. It involves taking a row of a matrix, to form a vector, forming a subvector, and changing the valid index range for the vector.

```
// loop form of the code
for (i=1;i<nyrs;i++)
{
  for (j=1;j<nages;j++)
  {
    N(i+1,j+1)=N(i,j)*S(i,j);
  }
}

// can only eliminate the inside loop
for (i=1;i<nyrs;i++)
{
  // ++ increments the index bounds by 1
  N(i+1)(2,nyrs)=++elem_prod(N(i)(1,nage-1),S(i)(1,nage-1));
}
```

Notice that N(i+1) is a vector object so that N(i+1)(2,nyrs) is a subvector of N(i). Another point is that elem_prod(N(i)(1,nage-1),S(i)(1,nage-1)) is a vector object with minimum valid index 1 and maximum valid index nyrs-1. The operator ++ applied to a subvector increments the valid index range by 1 so that it has the same range of valid index values as N(i+1)(2,nyrs). The operator -- would decrement the valid index range by 1.

The example contained in the file FOURD.TPL illustrates some aspects of the use of three and four dimensional arrays. There are now examples of the use of arrays up to dimension 7 in the documentation[2].

```
DATA_SECTION
  init_4darray d4(1,2,1,2,1,3,1,3)
  init_3darray d3(1,2,1,3,1,3)
PARAMETER_SECTION
  init_matrix M(1,3,1,3)
  4darray p4(1,2,1,2,1,3,2,3)
  objective_function_value f
PRELIMINARY_CALCS_SECTION
 for (int i=1;i<=3;i++)
 {
   M(i,i)=1;   // set M equal to the identity matrix to start
 }
PROCEDURE_SECTION
 for (int i=1;i<=2;i++)
 {
   for (int j=1;j<=2;j++)
   {
     // d4(i,j) is a 3x3 matrix -- d3(i) is a 3x3 matrix
     // d4(i,j)*M is matrix multiplication -- inv(M) is matrix inverse
```

[2]See the chapter on regime switching models for an example of the use of higher dimensional arrays.

```
      f+= norm2( d4(i,j)*M + d3(i)+ inv(M) );
    }
  }
REPORT_SECTION
  report << "Printout of a 4 dimensional array" << endl << endl;
  report << d4 << endl << endl;
  report << "Printout of a 3 dimensional array" << endl << endl;
  report << d3 << endl << endl;
```

In the DATA_SECTION you can use 3darrays, 4darrays, up to 7darrays and init_3darrays, init_4darrays up to init_7darrays. In the PARAMETER_SECTION you can use 3darrays, 4darrays, up to 7darrays and init_3darrays, init_4darrays up to init_5darrays at the time of writing.

If d4 is a 4darray then d4(i) is a three dimensional array and d4(i,j) is a matrix object so that d4(i,j)*M is matrix multiplication. Similarly if d3 is a 3darray then d3(i) is a matrix object so that d4(i,j)*M + d3(i) +inv(M) combines matrix multiplication, matrix inversion, and matrix addition.

The TOP_OF_MAIN section is intended to allow the programmer to insert any desired C++ code at the top of the main() function in the program. The code is copied literally from the template to the program. This section can be used to set the AUTODIF global variables (see the AUTODIF manual chapter on AUTODIF global variables.) The following code fragment will set these variables.

```
TOP_OF_MAIN_SECTION
  arrmblsize = 200000; // use instead of
                   // gradient_structure::set_ARRAY_MEMBLOCK_SIZE
  gradient_structure::set_GRADSTACK_BUFFER_SIZE(100000); // this may be incorrect in
                                        // the AUTODIF manual.
  gradient_structure::set_CMPDIF_BUFFER_SIZE(50000);
  gradient_structure::set_MAX_NVAR_OFFSET(500); // can have up to 500
                                              // independent variables
  gradient_structure::set_MAX_NUM_DEPENDENT_VARIABLES(500); // can have up to
                                           // 500 dependent variables
```

Note that within AD Model Builder one doesn't use the function gradient_structure::set_ARRAY_MEMBLOCK_SIZE to set the amount of memory available for variable arrays. Instead use the line of code arrmblsize = nnn; where nnn is the amount of memory desired.

The GLOBALS_SECTION is intended to allow the programmer to insert any desired C++ code before the main() function in the program. The code is copied literally from the template to the program. This enables the programmer to define global objects and to include include header files and user-defined functions into the generated C++ code.

Code in the between phases section is executed before each phase of the minimization. It is possible to carry out different actions which depend on which phase of the minimization is to begin by using a `swtich` statement (you can read about this in a book on C or C++) together with the `current_phase()` function.

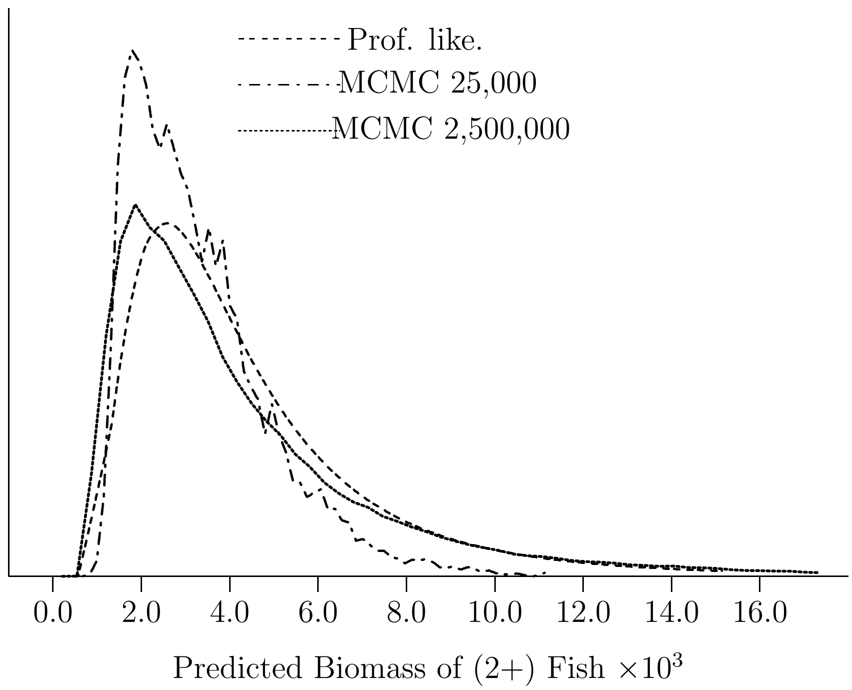
```
switch (current_phase()
{
case 1:
  // some action
  cout << "Before phase 1 minimization " << endl;
  break;
case 2: i
  // some action
  cout << "Before phase 2 minimization " << endl;
  break;
// ....
}
```

Chapter 2

Markov Chain Simulation

The reference for this chapter is *Bayesian Data Analysis* (chapter 11) by Gelman *et al.*

The Markov chain Monte Carlo method (MCMC) is a method for approximating the posterior distribution for parameters of interest in the Bayesian framework. This option is invoked by using the command line option `-mcmc N` where `N` is the number of simulations performed. You will proabably also want to include the option `-mcscale` which dynamically scales the covariance matrix until a reasonable acceptance rate is observed. You may also want to use the `-mcmult n` option which scales the initial covariances matrix if the initial values are so large that arithmetic errors occur. One advantage of AD Model Builder over some other implementations of MCMC is that the mode of the posterior distribution together with the hessian at the mode is available to use for the MCMC routine. This information is used to implement a version of the Hastings-Metropolis algorithm. Another advantage is that with AD Model Builder it is possible to calculate the profile likelihood for a parameter of interest and compare the distribution to the MCMC distribution for that parameter. A large discrepancy may indicate that one or both estimates are inadequate. If you wish to do more simulations (and to carry on from where the last one ended use the `-mcr` option. The following figure compares the profile likelihood for the projected biomass to the estimates produced by the MCMC method for different sample sizes (25,000 and 2,500,000 samples) for the `catage` example.



A report containing the observed distributions is produced in the file `root.hst`. All objects of type `sdreport` i.e `number`, `vector` or `matrix` are included. It is possible to save the results of every `n'th` simulation by using the `-mcsave n` option. Afterwords these values can be used by running the model with the `-mceval` option which will evaluate the `userfunction` once for every saved simulation value. At this time the function `mceval_phase()` will return the value true and can be used as a switch to perform desired calculations. The results are saved in a binary file `root.psv`. If you want to convert this file into ASCII see the next section.

AD Model Builder uses the hessian to produce an (almost) multivariate normal distribution for the Metropolis-Hastings algorithm. It is not exacly multivariate normal because the random vectors produced are modified to satisfy any bounds on the parameters.

There is also an option for using a fatter tailed distribution. This distribution is a mixture of the multivariate normal and a fat-tailed distribution. It is invoked with the `-mcgrope n` option where `n` is the amount of fat-tailed distribution in the mixture. Proabably a value of `n` between 0.05 and 0.10 is best.

Often the data which AD Model Builder needs to save are saved in the form of a binary file using the `uistream` and `uostream` classes. If these data consist of a series of vectors all of which have the same dimension they are often saved in this form where the dimension is saved at the top of the file ad the vectors are saved afterword. It may be useful to convert thes numbers into binary form so that they can be put into other programs such as spreadsheets.

the following code will read the contents of these binary files. YOu should call the program readbin.cpp. It should be a simple matter to modify this program for other uses.

```
#include <fvar.hpp>
/*  program to read a binary file (using ADMB's uistream and
    uostream stream classes)  of vectors of length n.
    It is assumed that the size n is stored at the top of
    the file. there is no information about any many vectors
    are stored so we must check for an eof after each read
    To use the program you type:

                 readbin filename
*/
void produce_comma_delimited_output(dvector& v)
{
  int i1=v.indexmin();
  int i2=v.indexmax();
  for (int i=i1;i<=i2;i++)
  {
    cout << v(i) << ",";
  }
  cout << endl;
}

main(int argc, char * argv[])
{
  if (argc < 2)
  {
    cerr << " Usage:   progname inputfilename" << endl;
    exit(1);
  }
  uistream uis = uistream(argv[1]);
  if (!uis)
  {
    cerr << " Error trying to open binary input file "
         <<  argv[1] << endl;
    exit(1);
  }
  int ndim;
  uis >> ndim;
  if (!uis)
  {
    cerr << " Error trying to read dimension of the vector"
            " from the top of the file "
         <<  argv[1] << endl;
    exit(1);
  }
  if (ndim <=0)
  {
    cerr << " Read invalid dimension for the vector"
            " from the top of the file "
         <<  argv[1] << " the number was " << ndim << endl;
    exit(1);
  }

  int nswitch;
  cout << " 1 to see all records" << endl
       << " 2 then after the prompts  n1 and  n2 to see all" << endl
```

```
     << " records between n1 and n2 inclusive" <<  endl
     << " 3 to see the dimension of the vector" << endl
     << " 4 to see how many vectors there are" << endl;
cin >> nswitch;
dvector rec(1,ndim);
int n1=0;
int n2=0;
int ii=0;
switch(nswitch)
{
case 2:
  cout << " Put in the number for the first record you want to see"
       << endl;
  cin >> n1;
  cout << " Put in the number for the second record you want to see"
       << endl;
  cin >> n2;
case 1:
  do
  {
    uis >> rec;
    if  (uis.eof()) break;
    if (!uis)
    {
      cerr << " Error trying to read vector number " << ii
           << " from file " <<  argv[1] << endl;
      exit(1);
    }
    ii++;
    if (!n1)
    {
      // comment out the one you don't want
      //cout << rec << endl;
      produce_comma_delimited_output(rec);
    }
    else
    {
      if (n1<=ii && ii<=n2)
      {
        // comment out the one you don't want
        //cout << rec << endl;
        produce_comma_delimited_output(rec);
      }
    }
  }
  while (1);
  break;
case 4:
  do
  {
    uis >> rec;
    if  (uis.eof()) break;
    if (!uis)
    {
      cerr << " Error trying to read vector number " << ii
           << " from file " <<  argv[1] << endl;
      exit(1);
    }
```

```
      ii++;
    }
    while (1);
    cout << " There are " << ii << " vectors" << endl;
    break;
  case 3:
    cout << " Dimension = "  << ndim  << endl;
  default:
    ;
  }
}
```

Chapter 3

A forestry model – estimating the size distribution of wildfires

This examples highlights two features of AD Model Builder, the use of a numerical integration routine within a statistical parameter estimation model and the use of the `ad_begin_funnel` mechanism to reduce the size of temporary file storage required. It also provides a performance comparison between AD Model Builder and Splus.

This problem investigates a model which predicts a relationship between the size and frequency of wildfires. It is assumed that the probability of observing a wildfire in size category i is given by P_i, where

$$\log(P_i) = \ln(S_i - S_{i+1}) - \ln(S(1)).$$

If f_i is the number of widfires observed to lie in size category i the log-likelihood function for the problem is given by

$$l(\tau, \nu, \beta, \sigma) = \sum_i f_i \left[\ln(S_i - S_{i+1}) - \ln(S(1)) \right] \tag{3.1}$$

where S_i is defined by the integral

$$S_i = \int_{-\infty}^{\infty} \exp\Big\{ -z^2/2 + \tau\Big(-1 + \exp(-\nu a_i^\beta \exp(\sigma z))\Big)\Big\} dz \tag{3.2}$$

The parameters τ, ν, β, and σ are functions of the parameters of the original model, and don't have a simple interpretation. Fitting the model to data involves maximizing the above log-likelihood (0.1). While the gradient can be calculated (in integral form), coding it is cumbersome. Numerically maximizing the log-likelihood without specifying the gradient is preferable.

The parameter β is related to the fractal dimension of the perimeter of the fire. One hypothesis of interest is that $\beta = 2/3$ which is related to hypotheses about the nature of the mechanism by which fires spread. The AD Model Builder code for the model follows.

```
DATA_SECTION
 int time0
 init_int nsteps
 init_int k
 init_vector a(1,k+1)
 init_vector freq(1,k)
 int a_index;
 number sum_freq
!! sum_freq=sum(freq);
PARAMETER_SECTION
  init_number log_tau
  init_number log_nu
  init_number log_beta(2)
  init_number log_sigma
  sdreport_number tau
  sdreport_number nu
  sdreport_number sigma
  sdreport_number beta
  vector S(1,k+1)
  objective_function_value f
INITIALIZATION_SECTION
  log_tau 0
  log_beta -.405465
  log_nu 0
  log_sigma -2
PROCEDURE_SECTION
  tau=exp(log_tau);
  nu=exp(log_nu);
  sigma=exp(log_sigma);
  beta=exp(log_beta);
   funnel_dvariable Integral;
   int i;
   for (i=1;i<=k+1;i++)
   {
     a_index=i;
     ad_begin_funnel();
     Integral=adromb(&model_parameters::h,-3.0,3.0,nsteps);
     S(i)=Integral;
   }
   f=0.0;
   for (i=1;i<=k;i++)
   {
     dvariable ff=0.0;
     // make the model stable for case when S(i)<=S(i+1)
     // we have to subrtract s(i+1) from S(i) first or roundoff will
     // do away with the 1.e-50.
     f-=freq(i)*log(1.e-50+(S(i)-S(i+1)));
     f+=ff;
   }
   f+=sum_freq*log(1.e-50+S(1));
FUNCTION dvariable h(const dvariable& z)
  dvariable tmp;
  tmp=exp(-.5*z*z + tau*(-1.+exp(-nu*pow(a(a_index),beta)*exp(sigma*z))) );
```

```
  return tmp;
REPORT_SECTION
  int * pt=NULL;
  report << " elapsed time = "  << time(pt)-time0 << " seconds" << endl;
  report << "nsteps = " << setprecision(10) <<  nsteps << endl;
  report << "f = " << setprecision(10) <<  f << endl;
  report << "a" << endl << a << endl;
  report << "freq" << endl << freq << endl;
  report << "S" << endl << S << endl;
  report << "S/S(1)" << endl << setfixed << setprecision(6) << S/S(1) << endl;
  report << "tau "  << tau << endl;
  report << "nu "  << nu << endl;
  report << "beta "  << beta << endl;
  report << "sigma "  << sigma << endl;
```

The statement

```
   Integral=adromb(&model_parameters::h,-3.0,3.0,nsteps);
```

invokes the numerical integration routine for the user-defined function `h`. The function must be defined in a `FUNCTION` subsection. It can have any name, must be defined to take a `const dvariable&` argument, and must return a `dvariable`. The values -3.0, 3.0 are the limits of integration (effectively $-\infty$, ∞ for this example). The integer argument nsteps determines how accurate the integration will be. Higher values of nsteps will be more accurate but greatly increase the amount of time necessary to fit the model. The basic strategy is to use a moderate value for nteps, such as 6, and then to increase this value to see if the parameter estimates change much.

```
FUNCTION dvariable h(const dvariable& z)
```

Numerical integration routines can be very computationally intensive, especially when they must be computed to great accuracy. Such computations will require a lot of temporary storage in AD Model Builder. Fortunately the output from such a routine is just one number, the value of the integral. In automatic differentiation terminology a long set of computations which produce just one number is known as a funnel. It is possbile to exploit the properties of such a funnel to greatly reduce the amount of temporary storage required. All that is necessary is to declare an object of type `funnel_dvariable` and to assign the results of the computation to it. At the beginning of the funnel a call to the function `ad_begin_funnel` is made. There is quite a bit of overhead associated with the funnel construction so it should not be used for very small calculations. However it is possible to put it in and test the program to see whether it runs more quickly or not. The following modifed code will produce exactly the same results, but without the funnel construction.

```
  dvariable Integral;   // change the definition of Integral
  int i;
  for (i=1;i<=k+1;i++)
```

```
{
  a_index=i;
  // ad_begin_funnel();  // commment out this line
  Integral=adromb(&model_parameters::h,-3.0,3.0,nsteps);
  S(i)=Integral;
}
```

If the funnel construction is used on a portion of code which is not a funnel, incorrect derivative values will be obtained. If this is suspected the funnel should be removed as in the above example and the model run again.

The following report shows the amount of time required to run the model with a fxied value of β for different values of the parameter `nsteps`. For practical perposes a vlaue of nsteps=8 gives enough accuracy so that the model could be fit in about 6 seconds.

```
elapsed time = 2 seconds nsteps = 6 f = 629.9846518
tau 9.851110 nu 8.913479 beta 0.666667 sigma 1.885570

elapsed time = 2 seconds nsteps = 7 f = 629.9851092
tau 9.850213 nu 8.835066 beta 0.666667 sigma 1.882967

elapsed time = 6 seconds nsteps = 8 f = 629.9851223
tau 9.850227 nu 8.836769 beta 0.666667 sigma 1.883024

elapsed time = 6 seconds nsteps = 9 f = 629.9851222
tau 9.850226 nu 8.836769 beta 0.666667 sigma 1.883024

elapsed time = 14 seconds nsteps = 10 f = 629.9851222
tau 9.850226 nu 8.836769 beta 0.666667 sigma 1.883024
```

The corresponding times when beta was estimated in an extra phase of the minimization are given here. It as apparent tat the model parameters become unstable when beta is being estimated. Twice the log-likelihood difference is $2(629.98-627.31) = 5.34$ which is significant

```
elapsed time = 3 seconds nsteps = 6 f = 627.2919906
tau 20.729183 nu 427.816375 beta 0.180225 sigma 2.499445

elapsed time = 6 seconds nsteps = 7 f = 627.2952716
tau 21.868971 nu 80914.970724 beta 0.170392 sigma 4.232237

elapsed time = 17 seconds nsteps = 8 f = 627.297021
tau 22.858629 nu 2326271883.421848 beta 0.164749 sigma 7.653068

elapsed time = 62 seconds nsteps = 9 f = 627.2993787
tau 23.771061 nu 1652877622661391616.000000 beta 0.161073 sigma 14.451510

elapsed time = 123 seconds nsteps = 10 f = 627.3106333
tau 23.116097 nu 49753858778.636856 beta 0.159364 sigma 8.663666

elapsed time = 244 seconds nsteps = 11 f = 627.310624
tau 23.115275 nu 49009470510.133156 beta 0.159369 sigma 8.658643
```

The Splus minimizing routine nlminb was used to fit the model. Fitting the three parameter model with Splus required approximately 280 seconds compared to 6 seconds with AD Model Builder, so that AD Model Builder was approximately 45 times faster for this simple problem.

For the four parameter problem with beta estimated, the SPLUS routine exited after fourteen minutes and 30 seconds, reporting false convergence with a function value of 627.338.

The data for the example is

```
a
 0.04 0.1 0.2 0.4 0.8 1.6 3.2 6.4 12.8 25.6 51.2 102.4 204.8
freq
 167 84 61 29 19 17 4 4 1 0 1 1
```

where the first line contains the bounds for the size catagories and the second line contains the number of observations in each size category. The Splus code with fixed beta for the example is

```
obj.20<-
function(xvec)
{
#Objective for maxn in NLMINB   NB vector argument
 - llik.20(xvec[1], xvec[2], xvec[3])
}
llik.20<-
function(logtau, lognu, logsigma)
{
        tau<-exp(logtau)
        nu<-exp(lognu)
        sigma<-exp(logsigma)
print(tau)
print(nu)
print(sigma)
        llik <- 0
        for(i in 1:(length(freq)+1)) {
           Int[i]<-S.20(xa[i], tau, nu, sigma)
}
        print(llik)
        for(i in 1:length(freq)) {
           llik <- llik + (freq[i] * (log(1.e-50+(Int[i]-Int[i+1]))
               -log(1.e-50+Int[1])))
        }
        llik
}
S.20<-
function(da, tau, nu, sigma)
{
        results <- integrate(intgnd.20, -3, 3, TAU = tau, NU = nu, SIGMA =
                sigma, A = da)
        if(results$message != "normal termination")
                ans <- results$message
        else ans <- results$integral
        ans
}
intgnd.20<-
```

```
function(z, A, TAU, NU, SIGMA)
{
```

$$\exp(-0.5 * z^2 + TAU * (-1 + \exp(-NU * A^2/3 * \exp(SIGMA * z))))$$

```
}
```

To run the example in Splus with the same initial values use the following values

```
logtau 0  lognu 0  logsigma -2
```

The vector xa should contain the 13 `a` values while the vector freq should contain the 12 observed frequencies.

Chapter 4

Economic Models – regime switching

An active field in macroeconomic modeling is the area of "regime switching". This is discussed in greater generality in Hamilton (1994, chapter 22)[1]. The code for the following example is based on the domain switching model taken from Hamilton (1989)[2]. This example is not ideal for exploiting AD Model Builder's greatest advantage, the ability to estimate parameters in models with a large number of independent variables. However it does illustrate the efficacy of the use of higher (up to seven dimensional) arrays in AD Model Builder.

For this model The observed quantities are the Y_t where

$$Y_t = a_0 + a_1 s_{ti} + Z_t \tag{4.1}$$

and the state variables Z_t satisfy the fourth order autoregressive relationship

$$Z_t = f_1 Z_{t-1} + f_2 Z_{t-2} + f_3 Z_{t-3} + f_4 Z_{t-4} + \epsilon_t \tag{4.2}$$

where the ϵ_t are independent, normally distributed random variables with mean 0 and standard deviation σ. These equations correspond to Hamilton's equations 4.3. The state variable s_{ti} is the realized value of a Markov process, S_t, whose evolution is described below. This coefficient takes on the value i when the system is in state i. In the current example there are two states so that s_t takes on one of the two values 0 or 1. We can solve 0.1 for the values of Z_t conditioned on the unknown value of the state at time t. Let z_{ti} be defined by

$$\begin{aligned} z_{i0} &= Y_t - a_0 \\ z_{t1} &= Y_t - a_0 - a_1 \end{aligned} \tag{4.3}$$

[1] Hamilton, James D. 1994. *Time Series Analysis*, Princeton, N.J.: Princeton University Press.

[2] A new approach to the economic analysis of nonstationary time series and the business cycle, *Econometrica*, **57(2)**:357-384.

Let (i, j, k, l, m) be a quintuplet of state values for the states at time $t, t-1, \ldots, t-4$. Define $e(t, i, j, k, l, m)$, the realized values of the random variables ϵ_t by

$$e(t, i, j, k, l, m) = Y_{ti} - f_1 z_{t-1,j} - f_2 z_{t-2,k} - f_3 z_{t-3,l} - f_4 z_{t-4,m}$$

Notice that we due to the lags we can only begin to calculate values for the $e(t, i, j, k, l, m)$ in time period 5. It is assumed that the states transitions are given by a Markov process with transition matrix $P = (p_{ij})$[3]Hamilton seems to index his matrices with the column index first in some cases. We use the row index first. Thus Hamilton's p_{ij} may correspond to our p_{ji}. If we are in state j at time t the probability of being in state i at time $t+1$ is p_{ij}.

If we consider the quintuple of the last 5 states to be the states of a new markov process then we can define the transition matrix for this process by

$$(i, j, k, l, m) \Rightarrow (0, i, j, k, l) \quad \text{with probability } p_{0i}$$

and

$$(i, j, k, l, m) \Rightarrow (1, i, j, k, l) \quad \text{with probability } p_{1i}$$

If $q(t-1, j, k, l, m, n)$ is the probability of being in state (j, k, l, m, n) at period $t-1$ the probability of being in state $q(t, i, j, k, l, m)$ at time period t is given by

$$q(t, i, j, l, , m) = \sum_n P_{ij} q(t-1, j, k, l, m, n)$$

In particular if

$$q_b(t, i, j, k, l, m)$$

is the probability of being in the state (i, j, l, m, n) before observing Y_t and $q_a(t-1, j, k, l, m, n)$ is the probability of being in the state (j, k, l, m, n) after observing Y_{t-1} then

$$q_b(t, i, j, k, l, m) = \textstyle\sum_n P_{ij} q_a(t-1j, k, l, m, n) \tag{4.4}$$

Let $Q(Y_t|(i, j, k, l, m), Y_{t-1}, Y_{t-2}, Y_{t-3}, Y_{t-4})$ be the conditional probability (or probability density) for Y_t given $S_t = i, S_{t-1} = j, S_{t-2} = k, S_{t-3} = l, S_{t-4} = m, Y_{t-1}, Y_{t-2}, Y_{t-3}, Y_{t-4}$. Then, ignoring a constant term which is irrelevant for the calculations,

$$Q(Y_t|(i, j, k, l, m), Y_{t-1}, Y_{t-2}, Y_{t-3}, Y_{t-4}) = \exp(-e(i, j, k, l, m)^2/2\sigma^2)/\sigma \tag{4.5}$$

Define $u(Y_t, i, j, k, l, m)$ by

$$u(Y_t, i, j, k, l, m) = \quad Q(Y_t|(i, j, k, l, m), Y_{t-1}, \ldots, Y_{t-4}) q_b(t, i, j, k, l, m) \tag{4.6}$$

Then $q_a(t, i_t, j, k, l, m)$ can be calculated from the relationship

$$q_a(t, i_t, j, k, l, m) = u(Y_t, i, j, k, l, m) / \textstyle\sum_{i,j,k,l,m} u(Y_t, i, j, k, l, m) \tag{4.7}$$

The log-likelihood function for the parameters can be calculated from the $u(Y_t, i, j, k, l, m)$. It is equal to

$$\textstyle\sum_t \log(\sum_{i,j,k,l,m} u(Y_t, i, j, k, l, m)) \tag{4.8}$$

The sums needed for the calculations in 0.9 can be saved from the calculations for 0.8).

[3]†††

The complete AD Model Builder template (TPL) code is in the file `ham4.tpl` The C++ (CPP) code produced from this is in the file `ham4.cpp` Here is the TPL code split up with comments.

```
DATA_SECTION
  init_number a1init   // read in the initial value of a1 with the data
  init_int nperiods1   // the number of observations
  int nperiods  // nperiods-1 after differencing
 !! nperiods=nperiods1-1;
  init_vector yraw(1,nperiods1)  //read in the observations
  vector y(1,nperiods)   // the differenced observations
 !! y=100.*(--log(yraw(2,nperiods1)) - log(yraw(1,nperiods)));
  int order
  int op1
 !! order=4; //order of the autoregressive process
 !! op1=order+1;
  int nstates  // the number of states (expansion and contraction)
 !! nstates=2;
```

The `DATA_SECTION` contains constant quantities or "data". This is in contrast to quantities which depend on parameters being estimated which go into the `PARAMETER_SECTION`. All quantities in the `PARAMETER_SECTION`with the `init_` prefix are initial data which must be read in from somewhere. By default they are read in from the file `ROOT.dat` (DAT file) where `ROOT` is the root part of the name of the program being run (in this case `ham4.exe`), so `ham4.dat`.

The first quantity is a number, `a1init` which will be used for initializing the value of `a1` in the program. This is a simple way to try different initial values for `a1` simply by modifying the input data file. Such procedures are often valuable to ensure that the correct global value of the objective function has been found. The second quantity `nperiods1` is the number of data points in the file. Notice that as soon as a quantity has been defined it is available to use for defining other quantities. The quantity `nperiod` does not have an `init_` before it so it will not be read in an must be calulated in terms of other quantities at some point. Since we want it now it is calculated immediately.

```
 !! nperiods=nperiods1-1;
```

The `!!` are used to insert any valid C++ code into the `DATA_SECTION` or `PARAMETER_SECTION` (see `LOCAL_CALCS`). This code will be executed verbatim (after the `!!` have been stripped off of course) at the appropriate time. The `init_vector yraw` is defined and give a size with indices going from `1` to `nperiods1`. The `nperiods1` data points will be read into `yraw` from the DAT file. The data are immediately transformed and the resulting `nperiods` data point are put into `y`.

```
PARAMETER_SECTION
  init_vector f(1,order,1)  // coefficients for the atuoregressive
                            // process
  init_bounded_matrix Pcoff(0,nstates-1,0,nstates-1,.01,.99,2)
        // determines the transition matrix for the markov process
  init_number a0(5)  // equation 4.3 in Hamilton (1989)
  init_bounded_number a1(0.0,10.0,4);
```

```
!! if (a0==0.0) a1=a1init;  // set initial value for a1 as specified
                     // in the top of the file nham4.dat
 init_bounded_number smult(0.01,1,3)  // used in computing sigma
 matrix z(1,nperiods,0,1)  // computed via equation 4.3 in
                          // Hamilton (1989)
 matrix qbefore(op1,nperiods,0,1);  // prob. of being in state before
 matrix qafter(op1,nperiods,0,1); // and after observing y(t)
 number sigma // variance of epsilon(t) in equation 4.3
 number var  // square of sigma
 sdreport_matrix P(0,nstates-1,0,nstates-1);
 number ff1;
 vector qb1(0,1);
 matrix qb2(0,1,0,1);
 3darray qb3(0,1,0,1,0,1);
 4darray qb4(0,1,0,1,0,1,0,1);
 6darray qb(op1,nperiods,0,1,0,1,0,1,0,1,0,1);
 6darray qa(op1,nperiods,0,1,0,1,0,1,0,1,0,1);
 6darray eps(op1,nperiods,0,1,0,1,0,1,0,1,0,1);
 6darray eps2(op1,nperiods,0,1,0,1,0,1,0,1,0,1);
 6darray prob(op1,nperiods,0,1,0,1,0,1,0,1,0,1);
 objective_function_value ff;
```

The PARAMETER_SECTION describes the parameters of the model, that is, the quantities to be estimated. Quantities which which have the prefix init_ are akin to the independent variables from which the log-likelihood function (or more generally any objective function) can be calculated. Other objects are dependent variables which must be calculated from the independent variables. The default behaviour of AD Model Builder is to read in initial parameter values for the parameters from a PAR file if it finds one. Otherwise they are given default values consistent with their type. The quantity f is a vector of four coefficents for the autoregressive process. Pcoff is a 2×2 matrix which is used to parameterize The transition matrix P for the Markov process. Its values are restricted to lie between .01 and 0.99. smult is a number used to parameterize sigma and var (which is the variance) as a multiple of the mean squared residuals. This reparameterization undimensionalizes the calculation and is a good technique to employ for nonlinear modeling in general. The transition matrix P is defined to be of type sdreport_matrix so that the standard deviation estimates for its members will be included in the standard deviation report contained in the STD file. To date AD Model Builder suports up to seven dimensional arrays. For historical reasons one and two dimensional arrays are referred to as vector and matrix. This becomes a bit difficult for higer dimensional arrays so they are simply referred to as 3darray,4darray,...,7darray.

```
PROCEDURE_SECTION
  P=Pcoff;
  dvar_vector ssum=colsum(P);  // form a vector whose elements are the
                           // sums of the columns of P
  ff+=norm2(log(ssum)); // this is a penalty so that the hessian will
                       // not be singular and the coefficients of P
                       // will be well defined
  // normalize the transition matrix P so its columns sum to 1
  int j;
  for (j=0;j<=nstates-1;j++)
  {
```

```
  for (int i=0;i<=nstates-1;i++)
  {
    P(i,j)/=ssum(j);
  }
}

// get z into a useful format
dvar_matrix ztrans(0,1,1,nperiods);
ztrans(0)=y-a0;
ztrans(1)=y-a0-a1;
z=trans(ztrans);
int t,i,k,l,m,n;

qb1(0)=(1.0-P(1,1))/(2.0-P(0,0)-P(1,1)); // unconditional distribution
qb1(1)=1.0-qb1(0);

// for periods 2 through 4 there are no observations to condition
// the state distributions on so we use the unconditional distributions
// obtained by multiplying by the transition matrix P.
for (i=0;i<=1;i++) {
  for (j=0;j<=1;j++) qb2(i,j)=P(i,j)*qb1(j);
}

for (i=0;i<=1;i++) {
  for (j=0;j<=1;j++) {
    for (k=0;k<=1;k++) qb3(i,j,k)=P(i,j)*qb2(j,k);
  }
}

for (i=0;i<=1;i++) {
  for (j=0;j<=1;j++) {
    for (k=0;k<=1;k++) {
      for (l=0;l<=1;l++) qb4(i,j,k,l)=P(i,j)*qb3(j,k,l);
    }
  }
}

// qb(5) is the probabilibility of being in one of 32
// states (32=2x2x2x2x2) in periods 5,4,3,2,1 before observing
// y(5)
for (i=0;i<=1;i++) {
  for (j=0;j<=1;j++) {
    for (k=0;k<=1;k++) {
      for (l=0;l<=1;l++) {
        for (m=0;m<=1;m++) qb(op1,i,j,k,l,m)=P(i,j)*qb4(j,k,l,m);
      }
    }
  }
}
// now calculate the realized values for epsilon for all
// possible combinations of states
for (t=op1;t<=nperiods;t++) {
  for (i=0;i<=1;i++) {
    for (j=0;j<=1;j++) {
      for (k=0;k<=1;k++) {
        for (l=0;l<=1;l++) {
          for (m=0;m<=1;m++) {
            eps(t,i,j,k,l,m)=z(t,i)-phi(z(t-1,j),
```

```
              z(t-2,k),z(t-3,l),z(t-4,m),f);
            eps2(t,i,j,k,l,m)=square(eps(t,i,j,k,l,m));
          }
        }
      }
    }
  }
}
// calculate the mean squared "residuals" for use in
// "undimensionalized" parameterization of sigma
dvariable eps2sum=sum(eps2);
var=smult*eps2sum/(32.0*(nperiods-4));
sigma=sqrt(var);

for (t=op1;t<=nperiods;t++) {
  for (i=0;i<=1;i++) {
    for (j=0;j<=1;j++) {
      for (k=0;k<=1;k++)
        prob(t,i,j,k)=exp(eps2(t,i,j,k)/(-2.*var))/sigma;
    }
  }
}

for (i=0;i<=1;i++) {
  for (j=0;j<=1;j++) {
    for (k=0;k<=1;k++) {
      for (l=0;l<=1;l++) {
        for (m=0;m<=1;m++) qa(op1,i,j,k,l,m)= qb(op1,i,j,k,l,m)*
          prob(op1,i,j,k,l,m);
      }
    }
  }
}
ff1=0.0;
qbefore(op1,0)=sum(qb(op1,0));
qbefore(op1,1)=sum(qb(op1,1));
qafter(op1,0)=sum(qa(op1,0));
qafter(op1,1)=sum(qa(op1,1));
dvariable sumqa=sum(qafter(op1));
qa(op1)/=sumqa;
qafter(op1,0)/=sumqa;
qafter(op1,1)/=sumqa;
ff1-=log(1.e-50+sumqa);
for (t=op1+1;t<=nperiods;t++) { // notice that the t loop includes 2
  for (i=0;i<=1;i++) {      // i,j,k,l,m blocks
    for (j=0;j<=1;j++) {
      for (k=0;k<=1;k++) {
        for (l=0;l<=1;l++) {
          for (m=0;m<=1;m++) {
            qb(t,i,j,k,l,m).initialize();
            // here is where having 6 dimensional arrays makes the
            // formula for moving the state distributions form period
            // t-1 to period t easy to program and understand.
            // Throw away  n and accumulate its two values into next
            // time period after multiplying by transition matrix P
            for (n=0;n<=1;n++) qb(t,i,j,k,l,m)+=P(i,j)*qa(t-1,j,k,l,m,n);
          }
        }
```

```
          }
        }
      }
      for (i=0;i<=1;i++) {
        for (j=0;j<=1;j++) {
          for (k=0;k<=1;k++) {
            for (l=0;l<=1;l++) {
              for (m=0;m<=1;m++) qa(t,i,j,k,l,m)=qb(t,i,j,k,l,m)*
                    prob(t,i,j,k,l,m);
            }
          }
        }
      }
      qbefore(t,0)=sum(qb(t,0));
      qbefore(t,1)=sum(qb(t,1));
      qafter(t,0)=sum(qa(t,0));
      qafter(t,1)=sum(qa(t,1));
      dvariable sumqa=sum(qafter(t));
      qa(t)/=sumqa;
      qafter(t,0)/=sumqa;
      qafter(t,1)/=sumqa;
      ff1-=log(1.e-50+sumqa); // add small constant to avoid log(0)
    }
    ff+=ff1; //ff1 is minus the log-likelihood
    ff+=.1*norm2(f); // add small penalty to stabilize estimation
```

The PROCEDURE_SECTION is where the calculation of the objective function are carried out. First the transition matrix P is calculated from the Pcoff. The function colsum forms a vector whose elements are the column sums of the matrix. This is used to normalize P so that its columns sum to 1. A penalty is added to the objective function for the colum sums so that the hessian matrix with respect to the independent variables will not be singular. This does not affect the "statistical" properties of the parameters of interest. The matrix z is calculated using a transformed matrix because AD Model Builder deals with vector rows better than columns. The probability distribution for the states in period 1, qb1 is set equal to the uncondtional distribution for a Markov process in terms of its transition matrix, P, as discussed in Hamilton (1994). The transition matrix is used to compute the probability distribution of the states in periods $(2,1)$, $(3,2,1)$, $(4,3,2,1)$, and finally $(5,4,3,2,1)$. For the last quintuplet this is the probability distribution before observing y(5). The quantities eps in the code correspond to the possible realized values of the random variable ϵ. The quantities qa and qb correspond to q_a and q_b in the documentation. The sum function is defined for arrays of any dimension and simply forms the sum of all the components. In AD Model Builder if xx is an n dimensional array then x(i) is an n-1 dimensional array. So the statement

```
qbefore(t,0)=sum(qb(t,0));
```

takes the sum of the probabilities for the sixteen quintuples of states at time period t through t-4 for which the state at time period t is 0. These are used in the REPORT_SECTION to write out a report of the estimated state probabilities at time period t before and after observing y(t).

```
REPORT_SECTION
  dvar_matrix out(1,2,op1,nperiods);
  dvar_matrix out1(1,1,op1,nperiods);
  out(1)=trans(qbefore)(1);
  out(2)=trans(qafter)(1);
  {
    ofstream ofs("qbefore.rep");
    out1(1)=trans(qbefore)(0);
    ofs << trans(out1)<< endl;
  }
  {
    ofstream ofs("qafter.rep");
    out1(1)=trans(qafter)(0);
    ofs << trans(out1) << endl;
  }
  report << "#qbefore    qafter" <<  endl;
  report << setfixed << setprecision(3) << setw(7) << trans(out) << endl;
```

The `REPORT_SECTION` is used to report any result in a manner not already carried out by the models default behaviour. The probabilites of being in state 0 before and after observing `y(t)` are printed into the files `qbefore.rep` and `qafter.rep`. These vectors were stored in files so that they could be easily imported into graphing programs. The results are very similar to figure 1 in Hamilton (1989) as one might hope.

```
RUNTIME_SECTION
  maximum_function_evaluations 20000
  convergence_criteria 1.e-6
```

The `maximum_function_evaluations 20000` will simply let the program run a long time by setting the maximum number of function evaluations in the function minimizer equal to 20,000 (nowhere near this many are actually needed.) The convergence_criteria 1.e-6 was needed becuase the default value of `1.e-4` caused the program to exit from the miniization before convergence had been achieved.

```
TOP_OF_MAIN_SECTION
  arrmblsize=500000;
  gradient_structure::set_GRADSTACK_BUFFER_SIZE(200000);
  gradient_structure::set_CMPDIF_BUFFER_SIZE(2100000);
```

The `TOP_OF_MAIN_SECTION` is for including code which will be included at the top of the `main()` function in the C++ program. Any desired legal code may be included. There are a number of common statements which are used to control aspects of AD Model Builder's performance. The statement `arrmblsize=500000;` reserves 500,000 bytes of memory for variable objects. If it is not large enough a message will be printed out at run time. See the index for references to more discussion of this matter. The statements `gradient_structure::set_GRADSTACK_BUFFER_SIZE(200000);` and `gradient_structure::set_CMPDIF_BUFFER_SIZE(2100000);` set the amount of memory that AD Model Builder reserves for variable objects. Setting these is a matter of tuning for optimum performance. If you have a lot of memory available making them larger may improve performance. However models will run without including these statements as long as there is enough memory for AD Model Builder's temporary files.

```
GLOBALS_SECTION
  #include <admodel.h>

  dvariable phi(const dvariable& a1,const dvariable& a2,const dvariable& a3,
    const dvariable& a4,const dvar_vector& f)
  {
    return   a1*f(1)+a2*f(2)+a3*f(3)+a4*f(4);
  }
```

The `GLOBALS_SECTION` is used to include statements at the top of the file containing the `CPP` program. This is generally where global declarations are made in C++, hence its name. However it may be used for any legal statments such as including header files for the users data structures etc. In this case it has been used to define the function `phi` which is used to simplify the code for the model's calculations. The header file `admodel.hpp` is included to define the `AUTODIF` structures used in the definition of the function. This header is automatically included near the top of the file, but this would be too late as `GLOBALS_SECTION` material is included first.

The parameter estimates for the initial parameters are written into a file `HAM4.PAR`. This is an ASCII file wich can be easily read. (The results are also stored in a binary file `HAM4.BAR` which can be used to restart the model with more accurate parameters estimates.)

```
# Objective function value = 60.8934
# f:
 0.0139989 -0.0569580 -0.246292 -0.212250
# Pcoff:
 0.754133 0.0955834
 0.245118 0.900333
# a0:
-0.357964
# a1:
1.52138
# smult:
0.281342
```

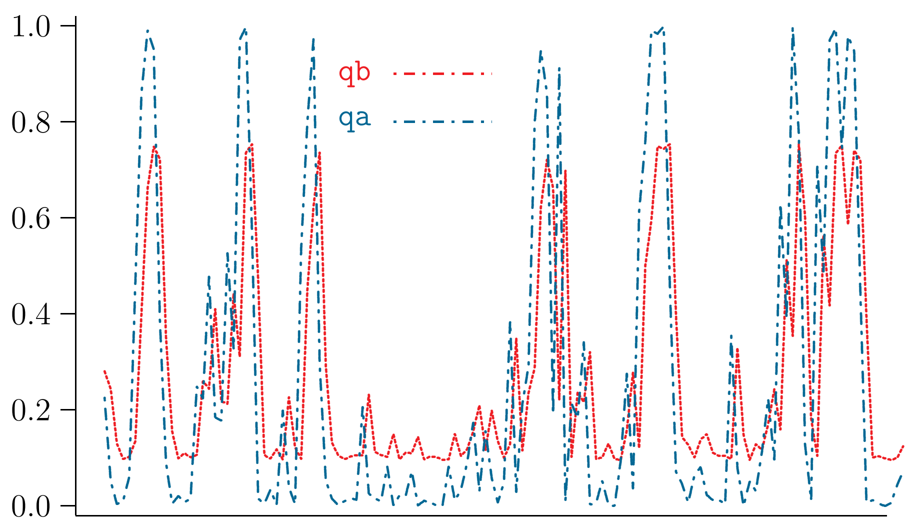
The estimates are almost identical to those reported in Hamilton (1989)[4] The first line reports the value of the log-likelihood function. This value can be used in hypothesis (likelihood-ratio) tests. the file `ham5.par`a for the fifth order autoregressive model fit to the data in Hamilton (1989) is shown below. there is one more parameter in this model. Twice the difference in the log-likelihood functions is $2(60.89 - 59.60) = 2.58$. For one extra parameter the 95% significance level is 3.84, the improvement in fit is not significant.

```
# Objective function value = 59.6039
# f:
 -0.0474771 -0.113829 -0.241966 -0.225535 -0.192585
# Pcoff:
```

[4] Our method for parameterizing the intial state probability distribution `qb1` is slightly different from Hamilton's which would explain the small discrepancy.

```
 0.779245 0.0951739
 0.219775 0.900719
# a0:
-0.271318
# a1:
1.46301
# smult:
0.259541
```

The plot of `qa` and `qb` demonstrates the extra information about the probability distribution of the current state contained in in the current value of `y(t)`.



Apriori and aposteriori Probabilitites of Being in State 0 in Period t

The standard deviation and correlation report for the model are in the file `ham4.cor` reproduced below.

```
index name  value    std dev    1     2     3     4     5     6     7     8     9    10    11    12    13    14   15

  1  f      1.39e-02 1.20e-01  1.00

  2  f     -5.69e-02 1.37e-01  0.33  1.00

  3  f     -2.46e-01 1.06e-01  0.33  0.29  1.00

  4  f     -2.12e-01 1.10e-01  0.43  0.26  0.17  1.00

  5  Pcoff  7.54e-01 5.39e-01  0.00  0.04  0.01  0.00  1.00

  6  Pcoff  9.55e-02 7.58e-02  0.04  0.05  0.02  0.03 -0.04  1.00

  7  Pcoff  2.45e-01 1.97e-01 -0.01 -0.11 -0.03 -0.01  0.77  0.04  1.00

  8  Pcoff  9.00e-01 6.20e-01 -0.00 -0.00 -0.00 -0.00  0.00  0.83 -0.00  1.00

  9  a0    -3.57e-01 2.65e-01  0.27  0.56  0.25  0.21  0.08  0.07 -0.23 -0.00  1.00

 10  a1     1.52e+00 2.63e-01 -0.31 -0.57 -0.29 -0.25 -0.07 -0.04  0.21  0.00 -0.96  1.00

 11  smult  2.81e-01 1.25e-01  0.54  0.69  0.48  0.45  0.06  0.05 -0.17 -0.00  0.82 -0.84  1.00

 12  P      7.54e-01 9.65e-02  0.02  0.24  0.07  0.03  0.17 -0.08 -0.48  0.00  0.47 -0.44  0.36  1.00

 13  P      9.59e-02 3.77e-02  0.09  0.10  0.04  0.06 -0.02  0.49  0.08 -0.05  0.14 -0.09  0.11 -0.16  1.00

 14  P      2.45e-01 9.65e-02 -0.02 -0.24 -0.07 -0.03 -0.17  0.08  0.48 -0.00 -0.47  0.44 -0.36 -1.00  0.16  1.00

 15  P      9.04e-01 3.77e-02 -0.09 -0.10 -0.04 -0.06  0.02 -0.49 -0.08  0.05 -0.14  0.09 -0.11  0.16 -1.00 -0.16 1.00
```

Hamilton (1989, page 372) remarks that investigating higher order autoregressive processes might be a fruitful area of research. The form of the model is. The first extension of the model is a fifth order autoregressive process.

$$Y_t = a_0 + a_1 s_{ti} + Z_t \tag{4.9}$$

and the state variables Z_t satisfy the fourth order autoregressive relationship

$$Z_t = f_1 Z_{t-1} + f_2 Z_{t-2} + f_3 Z_{t-3} + f_4 Z_{t-4} + f_5 Z_{t-5} + \epsilon_t \tag{4.10}$$

which extend equations 0.1 and 0.2. The `TPL` file `ham5.tpl` for the fifth order autoregressive model is reproduced here. By employing higher dimensional arrays the conversion of the `TPL` file from a fourth order autoregressive process to a fifth order one is largely formal. An experienced AD Model Builder user can carry out the modifications in under 1 hour. Places where modifications were made were tagged with the comment tt //!!5.

```
DATA_SECTION
  init_number a1init   // read in the initial value of a1 with the data
  init_int nperiods1   // the number of observations
```

```
  int nperiods  // nperiods-1 after differencing
 !! nperiods=nperiods1-1;
  init_vector yraw(1,nperiods1)  //read in the observations
  vector y(1,nperiods)   // the differenced observations
 !! y=100.*(--log(yraw(2,nperiods1)) - log(yraw(1,nperiods)));
  int order
  int op1
 !! order=5; // !!5 order of the autoregressive process
 !! op1=order+1;
  int nstates  // the number of states (expansion and contraction)
 !! nstates=2;
PARAMETER_SECTION
  init_vector f(1,order,1)  // coefficients for the atuoregressive
                            // process
  init_bounded_matrix Pcoff(0,nstates-1,0,nstates-1,.01,.99,2)
        // determines the transition matrix for the markov process
  init_number a0(5)  // equation 4.3 in Hamilton (1989)
  init_bounded_number a1(0.0,10.0,4);
 !! if (a0==0.0) a1=a1init;  // set initial value for a1 as specified
                     // in the top of the file nham4.dat
  init_bounded_number smult(0.01,1,3)  // used in computing sigma
  matrix z(1,nperiods,0,1)  // computed via equation 4.3 in
                          // Hamilton (1989)
  matrix qbefore(op1,nperiods,0,1);  // prob. of being in state before
  matrix qafter(op1,nperiods,0,1); // and after observing y(t)
  number sigma // variance of epsilon(t) in equation 4.3
  number var  // square of sigma
  sdreport_matrix P(0,nstates-1,0,nstates-1);
  number ff1;
  vector qb1(0,1);
  matrix qb2(0,1,0,1);
  3darray qb3(0,1,0,1,0,1);
  4darray qb4(0,1,0,1,0,1,0,1);
  5darray qb5(0,1,0,1,0,1,0,1,0,1); // !!5
  7darray qb(op1,nperiods,0,1,0,1,0,1,0,1,0,1,0,1);
  7darray qa(op1,nperiods,0,1,0,1,0,1,0,1,0,1,0,1);
  7darray eps(op1,nperiods,0,1,0,1,0,1,0,1,0,1,0,1);
  7darray eps2(op1,nperiods,0,1,0,1,0,1,0,1,0,1,0,1);
  7darray prob(op1,nperiods,0,1,0,1,0,1,0,1,0,1,0,1);
  objective_function_value ff;
PROCEDURE_SECTION
  P=Pcoff;
  dvar_vector ssum=colsum(P);  // forma a vector whose elements are the
                         // sums of the columns of P
  ff+=norm2(log(ssum)); // this is a penalty so that the hessian will
                      // not be singular and the coefficients of P
                      // will be well defined
  // normalize the transition matrix P so its columns sum to 1
  int j;
  for (j=0;j<=nstates-1;j++)
  {
    for (int i=0;i<=nstates-1;i++)
    {
      P(i,j)/=ssum(j);
    }
  }

  dvar_matrix ztrans(0,1,1,nperiods);
```

```
  ztrans(0)=y-a0;
  ztrans(1)=y-a0-a1;
  z=trans(ztrans);
  int t,i,k,l,m,n,p;

  qb1(0)=(1.0-P(1,1))/(2.0-P(0,0)-P(1,1)); // unconditional distribution
  qb1(1)=1.0-qb1(0);

  // for periods 2 through 4 there are no observations to condition
  // the state distributions on so we use the unconditional distributions
  // obtained by multiplying by the transition matrix P.
  for (i=0;i<=1;i++) {
    for (j=0;j<=1;j++) qb2(i,j)=P(i,j)*qb1(j);
  }

  for (i=0;i<=1;i++) {
    for (j=0;j<=1;j++) {
      for (k=0;k<=1;k++) qb3(i,j,k)=P(i,j)*qb2(j,k);
    }
  }

  for (i=0;i<=1;i++) {
    for (j=0;j<=1;j++) {
      for (k=0;k<=1;k++) {
        for (l=0;l<=1;l++) qb4(i,j,k,l)=P(i,j)*qb3(j,k,l);
      }
    }
  }
  // !!5
  for (i=0;i<=1;i++) {
    for (j=0;j<=1;j++) {
      for (k=0;k<=1;k++) {
        for (l=0;l<=1;l++) {
          for (m=0;m<=1;m++) qb5(i,j,k,l,m)=P(i,j)*qb4(j,k,l,m);
}
      }
    }
  }
  // qb(6) is the probabilibility of being in one of 64
  // states (64=2x2x2x2x2x2) in periods 5,4,3,2,1 before observing
  // y(6)
  for (i=0;i<=1;i++) {
    for (j=0;j<=1;j++) {
      for (k=0;k<=1;k++) {
        for (l=0;l<=1;l++) {
          for (m=0;m<=1;m++) { // !!5
            for (n=0;n<=1;n++) qb(op1,i,j,k,l,m,n)=P(i,j)*qb5(j,k,l,m,n);
          }
}
      }
    }
  }
  // now calculate the realized values for epsilon for all
  // possible combinations of states
  for (t=op1;t<=nperiods;t++) {
    for (i=0;i<=1;i++) {
      for (j=0;j<=1;j++) {
        for (k=0;k<=1;k++) {
```

```
          for (l=0;l<=1;l++) {
            for (m=0;m<=1;m++) {
              for (n=0;n<=1;n++) { // !!5
        eps(t,i,j,k,l,m,n)=z(t,i)-phi(z(t-1,j),
          z(t-2,k),z(t-3,l),z(t-4,m),z(t-5,n),f);
                eps2(t,i,j,k,l,m,n)=square(eps(t,i,j,k,l,m,n));
              }
    }
  }
}
      }
    }
  }
  // calculate the mean squared "residuals" for use in
  // "undimensionalized" parameterization of sigma
  dvariable eps2sum=sum(eps2);
  var=smult*eps2sum/(64.0*(nperiods-4));  //!!5
  sigma=sqrt(var);

  for (t=op1;t<=nperiods;t++) {
    for (i=0;i<=1;i++) {
      for (j=0;j<=1;j++) {
        for (k=0;k<=1;k++) {
          for (l=0;l<=1;l++)  //!!5
    prob(t,i,j,k,l)=exp(eps2(t,i,j,k,l)/(-2.*var))/sigma;
        }
      }
    }
  }

  for (i=0;i<=1;i++) {
    for (j=0;j<=1;j++) {
      for (k=0;k<=1;k++) {
        for (l=0;l<=1;l++) {
          for (m=0;m<=1;m++) {
            for (n=0;n<=1;n++) qa(op1,i,j,k,l,m,n)= qb(op1,i,j,k,l,m,n)*
      prob(op1,i,j,k,l,m,n);
          }
        }
      }
    }
  }
  ff1=0.0;
  qbefore(op1,0)=sum(qb(op1,0));
  qbefore(op1,1)=sum(qb(op1,1));
  qafter(op1,0)=sum(qa(op1,0));
  qafter(op1,1)=sum(qa(op1,1));
  dvariable sumqa=sum(qafter(op1));
  qa(op1)/=sumqa;
  qafter(op1,0)/=sumqa;
  qafter(op1,1)/=sumqa;
  ff1-=log(1.e-50+sumqa);
  for (t=op1+1;t<=nperiods;t++) { // notice that the t loop includes 2
    for (i=0;i<=1;i++) {      // i,j,k,l,m blocks
      for (j=0;j<=1;j++) {
        for (k=0;k<=1;k++) {
          for (l=0;l<=1;l++) {
            for (m=0;m<=1;m++) {
```

```
              for (n=0;n<=1;n++) { //!!5
                qb(t,i,j,k,l,m,n).initialize();
        // here is where having 6 dimensional arrays makes the
        // formula for moving the state distributions form period
        // t-1 to period t easy to program and understand.
        // Throw away  n and accumulate its two values into next
        // time period after multiplying by transition matrix P
                for (p=0;p<=1;p++) qb(t,i,j,k,l,m,n)+=P(i,j)*
  qa(t-1,j,k,l,m,n,p);
              }
            }
  }
}
      }
    }
    for (i=0;i<=1;i++) {
      for (j=0;j<=1;j++) {
        for (k=0;k<=1;k++) {
          for (l=0;l<=1;l++) {
            for (m=0;m<=1;m++) { // !!5
              for (n=0;n<=1;n++) qa(t,i,j,k,l,m,n)=qb(t,i,j,k,l,m,n)*
          prob(t,i,j,k,l,m,n);
            }
  }
}
      }
    }
    qbefore(t,0)=sum(qb(t,0));
    qbefore(t,1)=sum(qb(t,1));
    qafter(t,0)=sum(qa(t,0));
    qafter(t,1)=sum(qa(t,1));
    dvariable sumqa=sum(qafter(t));
    qa(t)/=sumqa;
    qafter(t,0)/=sumqa;
    qafter(t,1)/=sumqa;
    ff1-=log(1.e-50+sumqa);
  }
  ff+=ff1;
  ff+=.1*norm2(f);
REPORT_SECTION
  dvar_matrix out(1,2,op1,nperiods);
  out(1)=trans(qbefore)(1);
  out(2)=trans(qafter)(1);
  {
    ofstream ofs("qbefore4.tex");
    for (int t=5;t<=nperiods;t++)
    {
      ofs << (t-4)/100. << " " << qbefore(t,0) << endl;
    }
  }
  {
    ofstream ofs("qafter4.tex");
    for (int t=5;t<=nperiods;t++)
    {
      ofs << (t-4)/100. << " " << qafter(t,0) << endl;
    }
  }
  report << "#qbefore    qafter" <<  endl;
```

```
  report << setfixed << setprecision(3) << setw(7) << trans(out) << endl;
RUNTIME_SECTION
  maximum_function_evaluations 20000
  convergence_criteria 1.e-6
TOP_OF_MAIN_SECTION
  arrmblsize=500000;
  gradient_structure::set_GRADSTACK_BUFFER_SIZE(400000);
  gradient_structure::set_CMPDIF_BUFFER_SIZE(2100000);
  gradient_structure::set_MAX_NVAR_OFFSET(500);
GLOBALS_SECTION
  #include <fvar.hpp>
   // !!5
  dvariable phi(const dvariable& a1,const dvariable& a2,const dvariable& a3,
    const dvariable& a4,const dvariable& a5,const dvar_vector& f)
  {
    return   a1*f(1)+a2*f(2)+a3*f(3)+a4*f(4)+a5*f(5);
  }
```

Chapter 5

Econometric Models – simultaneous equations

For each t, $1 \le t \le T$ let y_t be an n dimensional vector and x_t be an n dimensional vector. Let B and Γ be $(n \times n)$ and $(n \times m)$ matrices and suppose that the relationship

$$By_t + \Gamma x_t = u_t$$

holds where the u_t are n dimensional random vectors of disturbances. The y_t are the endogenous variables in the system. The x_t are predetermined variables in the sense that they are independent of u_t. Note that for autoregressive models the x_t may contain values of y_j for $j < i$. In general not all of the coefficients of B and Γ are estimable. Interesting cases have special structure which are determined by the particular parameterization of of B, Γ and D. In particular it is generally assumed that $B_{ii} = 1$ for $1 \le i \le n$ and that B^{-1} exists.

Assume that for each t, u_t has a multivariate normal distribution with mean 0 and covariance matrix D. The log-likelihood function for B, Γ, and D is given by

$$L(B, \Gamma, D) = T/2\log(|B|^2) - T/2\log(|D|) - 1/2\textstyle\sum_{t=1}^{T}[By_t + \Gamma x_t]' D^{-1}[By_t + \Gamma x_t] \quad (5.1)$$

If there are no constraints on D the value of D which maximizes 5.1 can be solved for in terms of the other parameters and observations.This value $\hat{D}$ is given by

$$\hat{D} = 1/T\textstyle\sum_{t=1}^{T}[By_t + \Gamma x_t]'[By_t + \Gamma x_t] \quad (5.2)$$

substituting this value into (0.1) it can be shown that

$$1/2\sum_{t=1}^{T}[By_t + \Gamma x_t]'\hat{D}^{-1}[By_t + \Gamma x_t]$$

is a constant which can be ignored for the maximization so that equation 5.2

$$\tilde{L}(B,\Gamma) = T/2\log(|B|^2) - T/2\log(|\hat{D}|) \tag{5.3}$$

and the FIML estimates for B and Γ can be found by maximizing $\tilde{L}(B,\Gamma)$.

When there are constraints on the parameters of D then $\tilde{D}$ is no longer the maximum likelihood estimate for D so it is necessary to maximize 5.1 which is in general a numerically unstable problem. To successfully carry out the optimization it is necessary to obtain reasonable initial estimates for the parameters of B and Γ and to use a good method for parametrerizing D. Initial estimates for B and Γ can be obtained from ordinary least squares (OLS), that is find the values of B and Γ which minimize

$$\sum_{t=1}^{T} \|y_t - B^{-1}\Gamma x_t\|^2$$

To parameterize D note that $\hat{D}$ is an estimate of D so that we can parameterize D by

$$D = A\hat{D}A'$$

where A is a lower triangular matrix. If U is the choleski decomposition of D and $\hat{U}$ is the choleski decomposition of $\hat{D}$ then $A = \hat{U}^{-1}U$. It follows that A should be close to the identity matrix which is a good initial estimate for A.

To evaluate the model's performance simulated data were generated. The form of the model is

$$\begin{aligned}
y_{t1} + y_{t4} + y_{t5} - 2 + 0.45y_{t-1,1} &= u_{t1}\\
0.1y_{t1} + y_{t2} + 2.0y_{t5} - 1 - 0.6y_{t-1,1} + 0.25y_{t-1,2} &= u_{t2}\\
0.3y_{t1} - 0.2y_{t2} + y_{t3} + 1 &= u_{t3}\\
1.4y_{t2} - 3.1y_{t3} + y_{t4} + 1 &= u_{t4}\\
y_{t3} + y_{t4} + y_{t5} &= u_{t5}
\end{aligned} \tag{5.4}$$

with a covariance matrix

$$D = \begin{pmatrix} 0.512 & 0.32 & 0.256 & -1.28 & 0\\ 0.32 & 0.328 & -0.16 & -0.8 & 0\\ 0.256 & -0.16 & 1.728 & 0.16 & 0.8\\ -1.28 & -0.8 & 0.16 & 4.8 & 0.8\\ 0 & 0 & 0.8 & 0.8 & 0.928 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0.1 & 1 & 0 & 0 & 2 \\ 0.3 & -0.2 & 1 & 0 & 0 \\ 0 & 1.4 & -3.1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

$$\Gamma = \begin{pmatrix} -2 & 0.45 & 0 \\ -1 & -0.6 & 0.25 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

For this model $n = 5$, $m = 3$, and $x_t = (1, y_{t-1,1}, y_{t-1,2})$.

The eigenvalues of D are (0.006610.135830.49622.211625.44573) Having a small eigenvalue tends to produce simulated data that are difficult to analyze.

Forty time periods of data were generated by the simulator. The simulated y values are:

```
1.63252 3.00223 1.70246 1.34813 -1.12202
-2.87857 -7.72402 -3.88482 -1.24196 4.71024
1.12975 -7.92719 -3.85188 -2.33007 4.2984
1.48112 -2.25692 -1.15585 -2.26166 3.01061
-2.91887 -5.65015 -2.74198 -0.695815 4.51346
2.29715 0.524946 0.0268777 1.07624 -0.0898846
1.32854 6.17993 2.51613 1.67248 -2.39914
0.5661 -2.53219 -0.966376 0.00820516 1.76543
0.353591 -3.81146 -2.04431 -1.48574 2.67208
-2.22887 -2.33436 -1.66284 0.646399 2.26448
2.29896 -6.42238 -2.41106 -1.70633 3.50028
0.145878 1.85161 0.646578 -0.380955 0.709761
-0.779376 -7.60611 -3.79636 -1.63017 4.02658
-0.107371 -5.61361 -2.35816 -1.77719 3.67762
0.662221 -5.78832 -2.19632 -2.03071 4.37961
-0.570661 -7.42505 -3.28544 -2.94125 5.19422
0.0953742 -1.80617 -1.06915 -0.0320784 2.00018
-0.406986 -4.96143 -2.8084 -0.948902 3.1811
1.07219 -7.92608 -2.95484 -3.17022 5.12702
-0.495144 1.33611 -0.357291 -0.0260083 0.360653
-0.637878 -8.76117 -3.81638 -1.77116 4.82796
1.59717 -3.18571 -1.72708 -1.93975 2.79462
-1.13013 -2.20942 -1.30198 -0.603895 2.29486
-1.0103 -7.90106 -3.65303 -1.07367 4.66283
-1.02985 -3.00268 -1.63388 0.309992 2.97876
0.176882 -7.96282 -3.60299 -1.86289 4.86943
1.16904 -1.07952 -0.0969977 -0.74563 2.38399
-0.636119 -2.84841 -1.43676 -0.38474 2.51142
-1.72929 -5.39866 -2.51289 0.0978131 3.786
3.56302 3.79343 2.05613 1.43836 -1.2029
0.15806 -0.863882 -0.302119 -1.19212 1.38518
1.37323 -1.94413 -0.537631 -0.751294 1.42083
-0.404075 -8.53817 -3.58618 -3.33976 5.69071
0.362091 -5.78568 -2.46635 -2.33359 4.21899
-2.26158 -12.7075 -6.07426 -3.62455 8.49292
1.20438 -5.44629 -2.30249 -2.02905 3.82742
```

```
1.41463 -1.71734 -0.788698 -1.90306 2.2595
0.897156 1.28039 0.693579 0.318737 0.385857
-0.0330384 -1.55642 -0.189474 0.312385 1.57168
1.5747 0.827181 1.26032 0.813312 0.270432
```

For the x values the first time periods data $x_0 = (1, 1, 2)$ were supplied. The simulated x values are:

```
1 1 2
1 1.63252 3.00223
1 -2.87857 -7.72402
1 1.12975 -7.92719
1 1.48112 -2.25692
1 -2.91887 -5.65015
1 2.29715 0.524946
1 1.32854 6.17993
1 0.5661 -2.53219
1 0.353591 -3.81146
1 -2.22887 -2.33436
1 2.29896 -6.42238
1 0.145878 1.85161
1 -0.779376 -7.60611
1 -0.107371 -5.61361
1 0.662221 -5.78832
1 -0.570661 -7.42505
1 0.0953742 -1.80617
1 -0.406986 -4.96143
1 1.07219 -7.92608
1 -0.495144 1.33611
1 -0.637878 -8.76117
1 1.59717 -3.18571
1 -1.13013 -2.20942
1 -1.0103 -7.90106
1 -1.02985 -3.00268
1 0.176882 -7.96282
1 1.16904 -1.07952
1 -0.636119 -2.84841
1 -1.72929 -5.39866
1 3.56302 3.79343
1 0.15806 -0.863882
1 1.37323 -1.94413
1 -0.404075 -8.53817
1 0.362091 -5.78568
1 -2.26158 -12.7075
1 1.20438 -5.44629
1 1.41463 -1.71734
1 0.897156 1.28039
1 -0.0330384 -1.55642
```

For the estinmation process all the elements of the matrices B and Γ with value 0 were fixed at theere correct value. The FIML estimates for unconstrained covariance matrix D are given below.

$$B = \begin{pmatrix} 1 & 0 & 0 & 0.364395 & 0.364395 \\ 0.238411 & 1 & 0 & 0 & 1.37593 \\ 0.042875 & -0.330484 & 1 & 0 & 0 \\ 0 & 1.93026 & -4.90367 & 1 & 0 \\ 0 & 0 & 1.06339 & 1.09851 & 1 \end{pmatrix}$$

$$\Gamma = \begin{pmatrix} -0.917978 & 0.431377 & 0 \\ 0.473915 & -0.491422 & 0.19981 \\ 0.441089 & 0 & 0 \\ -0.126701 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$D = \begin{pmatrix} 0.938236 & 0.843743 & 0.506127 & -1.38383 & 0.314262 \\ 0.843743 & 1.55844 & 0.732235 & -0.591771 & 1.37195 \\ 0.506127 & 0.732235 & 0.430091 & -0.805866 & 0.62244 \\ -1.38383 & -0.591771 & -0.805866 & 4.18591 & -0.0363513 \\ 0.314262 & 1.37195 & 0.62244 & -0.0363513 & 1.75939 \end{pmatrix}$$

Since $D_{51} = 0$, and $D_{52} = 0$, these values were not well estimated by the unconstrained FIML procedure. Suppose that we know that their values should be 0 and that the value of $D_{55} \leq 1.0$. We incorporate this knowledge into the model by using penalty functions.

$$B = \begin{pmatrix} 1 & 0 & 0 & 0.594218 & 0.594218 \\ -0.321482 & 1 & 0 & 0 & 1.97809 \\ 0.0416175 & -0.365572 & 1 & 0 & 0 \\ 0 & 2.48468 & -6.10763 & 1 & 0 \\ 0 & 0 & 0.968057 & 1.02738 & 1 \end{pmatrix}$$

$$\Gamma = \begin{pmatrix} r - 1.3418 & 0.448342 & 0 \\ -1.14717 & -0.593754 & 0.183788 \\ 0.372114 & 0 & 0 \\ -0.0640792 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$D = \begin{pmatrix} 0.593424 & -0.325467 & 0.298836 & -1.43231 & 0.00401209 \\ -0.325467 & 0.650371 & -0.234747 & 0.441841 & 4.48456e-06 \\ 0.298836 & -0.234747 & 0.258147 & -0.900105 & 0.255262 \\ -1.43231 & 0.441841 & -0.900105 & 6.13619 & 0.180376 \\ 0.00401209 & 4.48456e-06 & 0.255262 & 0.180376 & 1.00262 \end{pmatrix}$$

This is the code in the TPL file split up by sections and commented on. The DATA_SECTION defines the data and some size aspects of the model structure. Objects which are prefixed by init_ will be read in from the data file.

```
  // This version incorporates constraints via penalty functions.
  //This is sample code to determine the parameters of a
  //sim,ultaneous equations model. The notation follows
  //that of Hamilton, Times Series Analysis, chapter 9.
  //the general form of the model is
  //
  //    By_t + Gamma x_t =u_t
  //
  //for t=1,...,T. The u_t have covariance matrix D.

DATA_SECTION
  init_int T // the number of observations
  init_int dimy // dimension of the vector of
              // endogenousavariables
  init_int dimx // dimension of the vector of
              // predetermined variables
  init_int num_Bpar // the number of parameters in
              // the elements of B to be estimated
  init_int num_Gpar // the number of parameters in
              // the elements of Gamma to be estimated
  init_matrix y(1,T,1,dimy) // the y_t
  init_matrix x(1,T,1,dimx) // the x_t
  int dimy1
 !! dimy1=dimy*(dimy+1)/2;  // size of symmetric matrix
```

The PARAMETER_SECTION describes the model's parameters. Objects which are prefixed by init_ are the independent variables of the model. For example Bpar is used to parameterize the nonzero elements of B. ch_Dpar is used to parameterize the lower triangular matrix of the correction from emp_D to the covariance matrix D. The minimization is done in a number of phases. The parameter kx is used to have a parameter which becomes active in phase 4, so that the minimization will take place in four stages. This parameter does not enter into the "real" part of the model.

```
PARAMETER_SECTION
  init_vector Bpar(1,num_Bpar)
  init_vector Gpar(1,num_Gpar)
  init_vector ch_Dpar(1,dimy1,2)
  matrix B(1,dimy,1,dimy)
  matrix D(1,dimy,1,dimy)  // the covariance matrix for the
                           // disturbances u_t
  matrix emp_D(1,dimy,1,dimy)  // the covariance matrix for the
  matrix Gamma(1,dimy,1,dimx)
  matrix ch_D(1,dimy,1,dimy)
  matrix z(1,T,1,dimy);
  objective_function_value f
  init_number kx(4);
```

The PROCEDURE_SECTION is where the models calculations are carried out. It is split up into a set of functions wheere the model specific pieces of code (different code for different models) are located. Finally the optimization for parameter estimation is calculated. Thsi depends on the phase of the optimzation procedure. A `switch` statment is used to vary the form of the objective function depending on the phase. The function `current_phase()` return the number of the current phase of the optimization. The function `last_phase()` returns 1 ("true") if the current phase is the last phase of the optimization. Quadratic penalty functions are put on the model's parameters and the penalty weights are decreased in subsequent phase. this proecedure helsp to stabilize the optimzation when several model parametes are highly correlated.

```
PROCEDURE_SECTION
  fill_B();    // this will vary from model to model
  fill_Gamma();  // this will vary from model to model

  calculate_empirical_copvariance_matrix();

  fill_D();  // this will vary from model to model

  calculate_constraints();  // this will vary from model to model

  int sgn;
  switch (current_phase())
  {
  case 1:
    {
      f+=0.1*norm2(Bpar);
      f+=0.1*norm2(Gpar);
      f+=0.1*norm2(ch_Dpar);
      dvar_matrix Binv=inv(B);
      for (int t=1;t<=T;t++)
      {
        dvar_vector z=y(t)+Binv*Gamma*x(t);
        f+=z*z;
      }
      break;
    }
  default:
    {
      f+= -0.5*T*log(square(det(B)))
       +0.5*T*ln_det(D,sgn);

      dvar_matrix Dinv=inv(D);
      dvariable f1=0.0;
      for (int t=1;t<=T;t++)
      {
        dvar_vector z=B*y(t)+Gamma*x(t);
        f1+=z*(Dinv*z);
      }
      f+=0.5*f1;
      if (!last_phase())
      {
        f+=0.1*norm2(Bpar);
        f+=0.1*norm2(Gpar);
        f+=0.1*norm2(ch_Dpar);
      }
      else
```

```
      {
        f+=0.001*norm2(Bpar);
        f+=0.001*norm2(Gpar);
        f+=0.001*norm2(ch_Dpar);
      }
    }
  }

  f+=square(kx);

FUNCTION fill_B
  B.initialize();
  for (int i=1;i<=dimy;i++)
    B(i,i)=1.0;

  // this is part of the special structure of the model
  int ii=1;
  B(2,1)=Bpar(1);
  B(3,1)=Bpar(2);

  B(3,2)=Bpar(3);
  B(4,2)=Bpar(4);

  B(4,3)=Bpar(5);
  B(5,3)=Bpar(6);

  B(5,4)=Bpar(7);
  B(1,4)=Bpar(8);

  B(1,5)=Bpar(8);
  B(2,5)=Bpar(9);


FUNCTION fill_Gamma
  Gamma.initialize();

  // this is the part of special structure of the model
  Gamma(1,1)=Gpar(1);
  Gamma(2,1)=Gpar(2);
  Gamma(3,1)=Gpar(3);
  Gamma(4,1)=Gpar(4);

  Gamma(1,2)=Gpar(5);
  Gamma(2,2)=Gpar(6);


  Gamma(2,3)=Gpar(7);


FUNCTION fill_D

  ch_D.initialize();
  // this is the special structure of the model
  int ii=1;
  for (int i=1;i<=dimy;i++)
  {
    for (int j=1;j<=i;j++)
      ch_D(i,j)=ch_Dpar(ii++);
```

```
    ch_D(i,i)+=1;
  }

  D=ch_D*emp_D*trans(ch_D); // so Ch_D is the Cholesky
                      // decomposition of D
FUNCTION calculate_empirical_copvariance_matrix

  for (int t=1;t<=T;t++)
    z(t)=B*y(t)+Gamma*x(t);

  emp_D=empirical_covariance(z);

FUNCTION  calculate_constraints

  double wt=1.0;
  switch (current_phase())
  {
  case 1:
    wt=1.0;
    break;
  case 2:
    wt=10.0;
    break;
  case 3:
    wt=100.0;
    break;
  default:
    wt=1000.0;
    break;
  }
  if (D(5,5)>1.0)
    f+=wt*square(D(5,5)-1.00);

  f+=wt*square(D(5,1));
  f+=wt*square(D(5,2));
```

The `REPORT_SECTION` prints out a report of the imodel's results.

```
REPORT_SECTION
  report << "B" << endl;
  report << B << endl;
  report << "Gamma" << endl;
  report << Gamma << endl;
  report << "D" << endl;
  report << D << endl;
  report << "eigenvalues of D" << endl;
  report << eigenvalues(D) << endl;
  report << "y" << endl;
  report << y << endl;
  report << "x" << endl;
  report << x << endl;
```

Chapter 6

The Kalman filter

The Kalman filter is a device for estimating parameters in a class of "time-series" like models which are put into state-space form. We have used the notation from Harvey, chapter 3. The general state space form is an multivariate time series

$$y_t = Z_t \alpha_t + d_t + \epsilon_t$$

where Z_t is an $N \times m$ matrix, d_t is an N dimensional vector, y_t is an N dimensional vector and ϵ_t is a set of serially uncorrelated N dimensional random vectors with mean 0 and correlation H_t. The elements of α_t are not observable but are assumed to be generated by a first order Markov process

$$\alpha_t = T_t \alpha_{t-1} + c_t + R_t \eta_t$$

where T_t is an $m \times m$ matrix, c_t is an $m \times 1$ vector, R_t is an $m \times g$ matrix and η_t is a $g \times 1$ vector of serially uncorrelated random vectors with mean 0 and covariance matrix H_t. The specification of the state space system is completed by two further assumptions: 1. The initial state vector α_0 has a mean of a_0 and a variance of P_0. 2. The random vectors ϵ_t and η_t are uncorrelated with each other and uncorrelated with the initial state.

In applications of the model many of the parameters Z_t, d_t, H_t, T_t, c_t, R_t, and Q_t may be independent of t in which case we will write them without the subscript. Also R may be the identity matrix in which case we will omit it.

As a simple example of such a model consider the (two dimensional) random walk observed with error example considered below.

$$\begin{aligned} \alpha_t &= \alpha_{t-1} + \eta_t \\ y_t &= \alpha_t + \epsilon_t \end{aligned} \tag{6.1}$$

For this model the following parameters are fixed.

$$T = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad R = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad d = (0,0) \quad c = (0,0)$$

while the covariance matrices Q and H are estimated. Their true values used for simulating the data were. The initial value of a is $(0, 0)$.

$$Q = \begin{pmatrix} 1 & 0.8 \\ 0.8 & 1 \end{pmatrix} \quad H = \begin{pmatrix} 3 & -2.5 \\ -2.5 & 3 \end{pmatrix}$$

For a moment go back to the general state space model. Given a_0, P_0 we recursively calculate the a number of quantities via the relationships

$$\begin{aligned}
a_{t|t-1} &= T_t a_{t-1} + c_t \\
P_{t|t-1} &= T_t P_{t-1} T_t' + R_t Q_t R_t' \\
v_t &= y_t - Z_t a_{t|t-1} - d_t \\
F_t &= Z_t P_{t|t-1} Z_t' + H_t \\
a_t &= a_{t|t-1} - P_{t|t-1} Z_t' F_t^{-1} v_t \\
P_t &= P_{t|t-1} - P_{t|t-1} Z_t' F_t^{-1} Z_t P_{t|t-1}
\end{aligned} \tag{6.2}$$

The log likelihood function for the models parameters is given by

$$\log L = -\frac{NT}{2}\log 2\pi - 0.5\sum_{t=1}^{T}\log|F_t| - 0.5\sum_{t=1}^{T} v_t F_t^{-1} v_t$$

The TPL file for the random walk kalman filter code follows

```
DATA_SECTION
  init_int nt
  init_int N
  init_int m
  int m1
  init_matrix Y(1,nt,1,N)
  matrix P0(1,m,1,m)
 !! P0.initialize();
 !! m1=m*(m+1)/2;
PARAMETER_SECTION
  init_bounded_vector Qcoff(1,m1,-10.,10.1)
  init_bounded_vector Hcoff(1,m1,-10.,10.1)
  init_vector a0(1,m)
  matrix T(1,m,1,m)
  matrix TT(1,m,1,m)
  vector d(1,N)
  vector c(1,m)
  matrix chQ(1,m,1,m)
  sdreport_matrix Q(1,m,1,m)
  matrix chH(1,N,1,N)
  sdreport_matrix H(1,N,1,N)
  matrix Z(1,N,1,m)
  matrix TZ(1,m,1,N)
  objective_function_value f
 LOCAL_CALCS
   d.initialize();
```

```
   c.initialize();
   Z.initialize();
   Z(1,1)=1; Z(2,2)=1;
   T.initialize();
   T(1,1)=1; T(2,2)=1;
   TZ=trans(Z);
   TT=trans(T);
PROCEDURE_SECTION
  setup_Q();
  setup_H();
  f+=kalman_filter();
  cout << " f = " << f << endl;

FUNCTION setup_Q
  chQ.initialize();
  int ii=1;
  for (int i=1;i<=m;i++)
    for (int j=1;j<=i;j++)
      chQ(i,j)=Qcoff(ii++);
  Q=chQ*trans(chQ);
FUNCTION setup_H
  chH.initialize();
  int ii=1;
  for (int i=1;i<=N;i++)
    for (int j=1;j<=i;j++)
      chH(i,j)=Hcoff(ii++);
  H=chH*trans(chH);

FUNCTION dvariable kalman_filter(void)
  dvar3_array P(0,nt,1,m,1,m);
  dvar3_array P1(1,nt,1,m,1,m);
  dvar3_array F(1,nt,1,N,1,N);
  dvar3_array Finv(1,nt,1,N,1,N);
  dvar_matrix Ptemp(1,m,1,m);
  dvar_matrix a(0,nt,1,m);
  dvar_matrix a1(1,nt,1,m);
  dvar_matrix v(1,nt,1,N);
  a(0)=a0;
  P(0)=P0;
  // This is the Kalman filter recursion. The objects tmp1
  // and tmp2 hold common calculations to optimize a bit
  int t;
  for (t=1;t<=nt;t++)
  {
    a1(t)=T*a(t-1)+c;
    P1(t)=T*P(t-1)*TT+Q;
    dvar_vector pred_y=Z*a1(t)+d;
    v(t)=Y(t)-pred_y;
    dvar_matrix tmp1=P1(t)*TZ;
    F(t)=Z*tmp1+H;
    Finv(t)=inv(F(t));
    dvar_matrix tmp2= tmp1*Finv(t);
    P(t)=P1(t)-tmp2*Z*P1(t);
    a(t)=a1(t)+tmp2*v(t);
  }
  int sgn=0;
  dvariable f=0.0;
  for (t=1;t<=nt;t++)
```

```
    f+=ln_det(F(t),sgn)+v(t)*Finv(t)*v(t);
  return f;
TOP_OF_MAIN_SECTION
  arrmblsize=20000000;
  gradient_structure::set_CMPDIF_BUFFER_SIZE(3000000);
  gradient_structure::set_GRADSTACK_BUFFER_SIZE(1000000);
```

This example was deliberatley not optimized as much as it could be in order to retain the flavour of the more general state space problem. For example since T is the identity matrix and c is the zero vector the line of code

```
a1(t)=T*a(t-1)+c;
```

reduces to

```
a1(t)=a(t-1);
```

The parameters being estimated are a_0, Q, and H.

To parameterize the covariance matrices the Choleski decomposition parameterization was used. This ensures that the covariance matrices are positive (semi) definite. The technique can be seen in the function `setup_Q`. The lower triangular matrix `ch_Q` is filled with parameters from a bounded vector

```
FUNCTION setup_Q
  chQ.initialize();
  int ii=1;
  for (int i=1;i<=m;i++)
    for (int j=1;j<=i;j++)
      chQ(i,j)=Qcoff(ii++);
  Q=chQ*trans(chQ);   // chQ is the choleski decomposition of Q
```

Notice that the bounded vector `Qcoff` has slightly asymmetric bounds. This is a simple way to ensure that its initial value is not identically 0 which would lead to a singular covariance matrix.

```
init_bounded_vector Qcoff(1,m1,-10.,10.1)
```

The model parameters, standard deviations and corellations are reproduced from the standadard ADMB report.

index	name	value	std dev	7	8	9	10	11	12	13	14	15	16
7	a0	-1.1682e+00	9.0191e-01	1.000									
8	a0	1.2218e+00	8.6442e-01	0.352	1.000								
9	Q	9.9468e-01	1.0862e-01	0.059	-0.006	1.000							
10	Q	7.8808e-01	7.8737e-02	0.038	0.028	0.683	1.000						
11	Q	7.8808e-01	7.8737e-02	0.038	0.028	0.683	1.000	1.000					
12	Q	8.7279e-01	9.6118e-02	-0.018	0.069	0.185	0.721	0.721	1.000				
13	H	3.1352e+00	1.8123e-01	-0.015	-0.007	-0.305	-0.136	-0.136	-0.018	1.000			
14	H	-2.7119e+00	1.4922e-01	-0.021	0.001	-0.102	-0.238	-0.238	-0.139	-0.692	1.000		
15	H	-2.7119e+00	1.4922e-01	-0.021	0.001	-0.102	-0.238	-0.238	-0.139	-0.692	1.000	1.000	
16	H	3.2264e+00	1.7936e-01	0.015	-0.029	-0.031	-0.121	-0.121	-0.249	0.370	-0.698	-0.698	1.000

The Choleski decomposition parameterization merely ensures that the matrix is positive semi-definite. By adding a small positive number to the diagonal elements one can ensure that the covariance matrix is positive definite and can speed up and improve the stability of the estimation. Of course what is meant by small will depend on the particular problem being considered. A modifed form of the routine `setup_Q` follows.

```
FUNCTION setup_Q
  int i;
  chQ.initialize();
  int ii=1;
  for (i=1;i<=m;i++)
    for (int j=1;j<=i;j++)
      chQ(i,j)=Qcoff(ii++);
  Q=chQ*trans(chQ);   // chQ is the choleski decomposition of Q
  for (i=1;i<=m;i++)
    Q(i,i)+=0.1;   // make Q positive definite
```

Performing this modification for the present model for both Q and H causes the program to converge about twice as fast.

Chapter 7

Applying the Laplace approximation to the Generalized Kalman Filter – with an application to Stochastic Volatility Models

Let y_i be an N dimensional multivariate time series for $i = 1, \ldots, n$ where y_i is a random vector with probability density function $p(y_i|\alpha_i)$. For each i, the α_i are random vectors which satisfy the condition

$$\alpha_i = T_i(\alpha_{i-1}, y_{i-1}) + \eta_i \tag{7.1}$$

where $\mu_{\eta_i} = 0$ and $\sigma^2_{\eta_i} = \sigma^2_\eta$.

Let $p(\alpha_1)$ be the probability density function for α_1 before y_1 is oberved. After observing y_1 we want to calculate the probability distribution of α_1 given y_1. This is given by

$$p(\alpha_1|y_1) = p(y_1|\alpha_1)p(\alpha_1)/p(y_1) \tag{7.2}$$

where

$$p(y_1) = \textstyle\int_{-\infty}^{\infty} p(y_1|\alpha_1)p(\alpha_1)\, d\alpha_1 \tag{7.3}$$

let $\phi(y_1, \alpha_1) = \log(p(y_1|\alpha_1)p(\alpha_1))$ Let $\hat{\alpha}_1(y_1) = \max_{\alpha_1}\{\phi(y_1, \alpha_1)\}$. Approximate ϕ by its second order taylor expansion in α_1 at $\hat{\alpha}_1$.

$$\phi(y_1, \alpha_1) \approx \phi(y_1, \hat{\alpha}_1) + D^2_{\alpha_1\alpha_1}\phi(y_1, \hat{\alpha}_1(y_1))(\alpha_1 - \hat{\alpha}_1(y), \alpha_1 - \hat{\alpha}_1(y)) \tag{7.4}$$

so that

$$p(y) \approx e^{\phi(y_1,\hat{\alpha}_1(y_1))} \textstyle\int_{-\infty}^{\infty} \exp\Big\{ -\big(-D^2_{\alpha_1\alpha_1}\phi(y_1, \hat{\alpha}_1(y_1))(\alpha_1 - \hat{\alpha}_1(y), \alpha_1 - \hat{\alpha}_1(y))\big)\Big\}\, d\alpha_1 \tag{7.5}$$

Making a change of variables and integrating we obtain

$$p(y_1) \approx e^{\phi(y_1,\hat{\alpha}_1(y_1))}(2\pi)^{n/2}|-D^2_{\alpha_1\alpha_1}\phi(y_1,\hat{\alpha}_1(y_1)))|^{-1/2} \tag{7.6}$$

This is the Laplace approximation to the integral in (7.3).

If the distribution of α_1 is (multivariate) normal and the distribution of $y_1|\alpha_1$ is multivariate normal then $\phi(y_1,\alpha_1)$ is a quadratic function of α_1 so that the Laplace approximation is exact. The advantage of the Laplace approximation is that it can be employed for non normal distributions.

To illustrate this advantage consider the simple one dimensional case where α_1 has a (univiariate) normal distribution with mean 0 and variance σ^2_α. Assume that the distribution of $y_1|\alpha_1$ is a fat-tailed distribution which is a mixture of 0.95 normal distribution and 0.05 cauchy distribution. Then

$$\phi(y_1,\alpha_1) = \log\Big[0.95\exp(-0.5(y_1-\alpha_1)^2/\sigma^2_y)+0.05\sqrt{2/\pi}/(1+(y-\alpha_1)^2/\sigma^2_y)\Big]-0.5\alpha_1^2/\sigma^2_\alpha+const \tag{7.7}$$

whereas if y_1 is assumed to have a normal distribution

$$\phi(y_1,\alpha_1) = -0.5(y_1-\alpha_1)^2/\sigma^2_y - 0.5\alpha_1^2/\sigma^2_\alpha \ +const \tag{7.8}$$

where *const* denotes some constant independent of α_1. There are two drawbacks to the use of $5.b$ If the value of y_1 is an outlier from the point of the normal model then it will have too much influence on the mode of the estimate of $p(\alpha_1|y_1)$. Also since the variance

$$\sigma^2_{\alpha_1|y_1} = D^2_{\alpha_1\alpha_1}\phi(y_1,\beta_i)\}^{-1} = \Big[1/\sigma^2_y + 1/\sigma^2_\alpha\Big]^{-1} \tag{7.9}$$

the variance is independent of the value of y_1 observed and $\sigma^2_{\alpha_1|y_1}$ will be underestimated. This is incorrect behaviour since if y_1 is an outlier it contains (almost) no information about the value of $p(\alpha_1|y_1)$ so that $p(\alpha_1|y_1)$ should be almost equal to $p(\alpha_1)$. The likelihood function based on $5a$ has the desired behaviour.

To calculate (7.6) it is necessary to maximize $\phi(y_1,\alpha_1)$ with respect to α_1 and to calculate its hessian matrix with respect to α_1.

For the maximization we employ the Newton-Raphson algorithm. Let $\beta_0 = \mu_{\alpha_1}$

$$\beta_{i+1} = \beta_i - \{D^2_{\alpha_1\alpha_1}\phi(y_1,\beta_i)\}^{-1}(D_{\alpha_1}\phi(y_1,\beta_i)) \tag{7.10}$$

This operation is carried out a fixed number r times and then $\hat{\alpha}_1(y_1) \approx \beta_r$. For "well behaved" problems the sequence β_i converges quadratically to $\hat{\alpha}_1(y_1)$. We approximate $p(\alpha_1|y_1)$ by a multivariate normal with

$$\begin{aligned}\mu_{\alpha_1|y_1} &= \beta_r \\ \sigma^2_{\alpha_1|y_1} &= \{-D^2_{\alpha_1\alpha_1}\phi(y_1,\beta_r)\}^{-1}\end{aligned}$$

and approximate $p(\alpha_2|y_1)$ by a multivariate normal with

$$\begin{aligned}\mu_{\alpha_2|y_1} &= T(\beta_r, y_1)\\ \sigma^2_{\alpha_2|y_1} &= D_{\alpha_1}T_1(\beta_r, y_1)\sigma^2_{\alpha_1|y_1}D_{\alpha_1}T_1(\beta_r, y_1)' + \sigma^2_\eta\end{aligned}$$

Now

$$p(y_2|y_1) = \int_{-\infty}^{\infty} p(y_2|\alpha_2)p(\alpha_2|y_1)\,d\alpha_2 \tag{7.11}$$

As above we maximize the integrand of (7.11) with respect to α_2 and use the Laplace approximation to the integral. This produces the sequence of conditional probabilities, $p(y_i|y_{i-1})$. The log-likelihood function for the observed sequence y_i is given by

$$\sum_{i=1}^n \log(p(y_i|y_{i-1})) \tag{7.12}$$

Although we have not explicitly shown them the conditional likelihood functions $p(y_i|y_{i-1})$ depend on a number of parameters. These parameters include the specification of T, other parameters in the probability density $p(y_i|\alpha_i)$ and parameters which determine σ^2_η. If we denote these parameters by θ and write $(p(y_i|y_{i-1},\theta))$ to indicate this dependence the log-likelihood function becomes

$$\sum_{i=1}^n \log(p(y_i|y_{i-1},\theta)) \tag{7.13}$$

the maximum likelihood estimates for the parameter vector θ are found by maximizing (7.13) with respect to θ.

The version of the stochastic volatility model presented here is from the paper Multivariate Stochastic Volatility Models: Estimation and a comparison with VGARCH Models by Danielsson.

It is assumed that y_i has a multivariate normal distribution with $\mu_{y_i} = 0$ and covariance matrix $\Omega_i(\alpha_i) = H_i(\alpha_i)RH_i(\alpha_i)$ where $H_i(\alpha_i)$ is an $m\times m$ diagonal matrix whose j'th element on the diagonal is given by $\exp(\alpha_{ij})/2$ where the α_{ij} satisfy the relationship

$$\alpha_i = w + \text{elem\_prod}(\delta, \alpha_{i-1}) + \text{elem\_prod}(\lambda_1, y_{i-1}) + \text{elem\_prod}(\lambda_2, |y_{i-1}|) + \eta_i \tag{7.14}$$

where η_i is a multivariate normal random variable with $\mu_{\eta_i} = 0$ and $\sigma^2_{\eta_i} = \sigma^2_\eta$. If u and v are two vectors with j'th component u_j and v_j elem_prod(i,v) is the vector with j'th componment u_jv_j. R is an $m\times m$ postive definite matrix satisfying $r_{jj} = 1$, that is a corellation matrix. Then

$$\log(p(y_i|\alpha_i)) = -0.5\log|\Omega_i(\alpha_i)| - 0.5y_i'\Omega_i(\alpha_i)^{-1}y_i \tag{7.15}$$

and the distribution of $\alpha_i|y_{i-1}$ is multivariate normal with mean vector and covariance matrix given by

$$\begin{aligned} \mu_{\alpha_i|y_{i-1}} &= w + \text{elem\_prod}(\delta, \mu_{\alpha_{i-1}|y_{i-1}}) + \text{elem\_prod}(\lambda, y_{i-1}) \\ \sigma^2_{\alpha_i|y_{i-1}} &= i \quad \text{diag}(\delta)\sigma^2_{\alpha_{i-1}|y_{i-1}}\text{diag}(\delta) + \sigma^2_\eta \end{aligned} \tag{7.16}$$

$\text{diag}(\delta)$ is the diagonal matrix whose diagonal is equal to the vector δ.

$$\begin{aligned} \log(p(y_i|\alpha_i)p(\alpha_i|y_{i-1})) &= -0.5\log|\Omega_i(\alpha_i)| - 0.5y_i'\Omega_i(\alpha_i)^{-1}y_i - 0.5\log|\sigma^2_{\alpha_i|y_{i-1}}| \\ &\quad -0.5(\alpha_i - \mu_{\alpha_i|y_{i-1}})'(\sigma^2_{\alpha_i|y_{i-1}})^{-1}(\alpha_i - \mu_{\alpha_i|y_{i-1}}) \end{aligned} \tag{7.17}$$

To perform the Newton-Raphson calculations it is necessary to calculate the first and second derivatives of expression (7.17) with respect to the parameter vector α. This is the most involved part of the calculations and will depend on the particular form of the model. In the present case the calculations are simplified by the fact that Ω_i only depends on α through the diagonal matrix $H(\alpha_i)$.

The probability density function $p(\alpha_1)$ is assumed to be multivariate normal with $\mu_{\alpha_1} = \theta_0$ and $\sigma^2_{\alpha_1} = 0$.

The data consist of the daily Mark/Dollar and Yen/dollar exchange rates and the US and Japaneese stock index data. There are 1301 time periods with some missing data. The missing data which are denoted by the impossibly large value of 10,000 were replaced with the average from the period before and after. They can howerever easily be estimated in the model is desired.

The model was fit with various combinations of the parameters and the log-likelihood was examined to investigate the improvement in fit due to the addition of the parameters.

Parameters in model	number of parameters	log-likelihood
$w, \delta, R, \sigma^2_\eta$	24	3774.7
$w, \delta, R, \sigma^2_\eta, \lambda_1$	28	3806.6
$w, \delta, R, \sigma^2_\eta, \lambda_1, \theta_0$	32	3808.6
$w, \delta, R, \sigma^2_\eta, \lambda_1, \theta_0, \lambda_2$	36	3811.2

The parameters θ_0 and λ_2 did not produce a significant improvement to the fit. λ_2 measures the asymmetry in the response of the variance to positive and negative shocks.

Here are the parameter estimates and their standard deviations for the model with $w, \delta, R, \sigma_\eta^2$, and λ_1.

```
index   name    value      std dev
   1   w(1)       -1.3749e-001 4.9434e-002
   2   w(2)       -6.5649e-001 1.6161e-001
   3   w(3)        3.1693e-002 1.0574e-002
   4   w(4)       -1.2973e-002 1.5375e-002
   5   lambda1(1)  1.5564e-001 4.9688e-002
   6   lambda1(2)  1.8647e-001 6.9525e-002
   7   lambda1(3)  -6.9265e-002 1.4158e-002
   8   lambda1(4)  -1.6689e-001 3.1626e-002
   9   delta(1)    8.2229e-001 4.6074e-002
  10   delta(1)    5.0848e-001 1.0785e-001
  11   delta(1)    9.5763e-001 1.4602e-002
  12   delta(1)    9.3610e-001 1.8812e-002
  29   R(1,1)      1.0000e+000 0.0000e+000
  30   R(1,2)      5.3821e-001 2.2883e-002
  31   R(1,3)      -7.1704e-002 2.9477e-002
  32   R(1,4)      -3.8796e-002 2.9278e-002
  33   R(2,1)      5.3821e-001 2.2883e-002
  34   R(2,2)      1.0000e+000  0.0000e+000
  35   R(2,3)      -1.2932e-001 2.9111e-002
  36   R(2,3)      -4.1466e-002 2.9468e-002
  37   R(3,1)      -7.1704e-002 2.9477e-002
  38   R(3,2)      -1.2932e-001 2.9111e-002
  39   R(3,3)      1.0000e+000  0.0000e+000
  40   R(1,4)      8.8811e-002 2.9085e-002
  41   R(4,1)      -3.8796e-002 2.9278e-002
  42   R(4,2)      -4.1466e-002 2.9468e-002
  43   R(4,3)      8.8811e-002 2.9085e-002
  44   R(4,4)      1.0000e+000  0.0000e+000
  45   Omega(1,1)  6.5973e-001 6.3099e-002
  46   Omega(1,2)  1.9827e-001 1.6129e-002
  47   Omega(1,3)  -1.3395e-001 5.4982e-002
  48   Omega(1,4)  -3.5161e-002 2.6676e-002
  49   Omega(2,1)  1.9827e-001 1.6129e-002
  50   Omega(2,2)  2.0570e-001 2.3994e-002
  51   Omega(2,3)  -1.3489e-001 3.2608e-002
  52   Omega(2,4)  -2.0985e-002 1.5016e-002
  53   Omega(3,1)  -1.3395e-001 5.4982e-002
  54   Omega(3,2)  -1.3489e-001 3.2608e-002
  55   Omega(3,3)  5.2895e+000 5.7872e-001
  56   Omega(3,4)  2.2791e-001 7.9318e-002
  57   Omega(4,1)  -3.5161e-002 2.6676e-002
  58   Omega(4,2)  -2.0985e-002 1.5016e-002
  59   Omega(4,3)  2.2791e-001 7.9318e-002
  60   Omega(4,4)  1.2451e+000 1.7043e-001
  61   Z(1,1)      2.3967e-001 7.4268e-002
  62   Z(1,2)      2.0711e-001 5.5599e-002
  63   Z(1,3)      3.8832e-002 1.8505e-002
  64   Z(1,4)      2.4097e-002 2.0344e-002
  65   Z(2,1)      2.0711e-001 5.5599e-002
  66   Z(2,2)      4.6309e-001 1.1143e-001
  67   Z(2,3)      3.4298e-002 2.3017e-002
  68   Z(2,4)      9.6831e-003 2.9999e-002
  69   Z(3,1)      3.8832e-002 1.8505e-002
  70   Z(3,2)      3.4298e-002 2.3017e-002
```



```
 71   Z(3,3)      3.9101e-002 1.6885e-002
 72   Z(3,4)      2.4602e-002 1.1053e-002
 73   Z(4,1)      2.4097e-002 2.0344e-002
 74   Z(4,2)      9.6831e-003 2.9999e-002
 75   Z(4,3)      2.4602e-002 1.1053e-002
 76   Z(4,4)      9.6109e-002 3.4268e-002
```

The AD Model Builder TPL file for the model is given below.

```
DATA_SECTION
  init_int ndim
  init_int nobs
  int ndim1
  int ndim2
 !! ndim1=ndim*(ndim+1)/2;
 !! ndim2=ndim*(ndim-1)/2;
  init_matrix Y(1,nobs,1,ndim)
 LOC_CALCS
  // replace missing values (10000) with the average of before and after.
  for (int i=2;i<nobs;i++)
    for (int j=1;j<=ndim;j++)
      if (Y(i,j)==10000)
      {
        int i2=i+1;
        do
        {
          if (Y(i2,j)==10000)
            i2++;
          else
            break;
        }
        while(1);
        Y(i,j)=(Y(i-1,j)+Y(i2,j))/2.;
        if (Y(i,j)>100.0)     // did this work
          cerr << " Y(i,j) too big " << Y(i,j) << endl;
      }
 END_CALCS

PARAMETER_SECTION
  matrix h_mean(1,nobs,1,ndim)
  3darray h_var(1,nobs,1,ndim,1,ndim)
  number ldR;
  init_vector theta0(1,ndim,3);
  vector lmin(1,nobs)
  init_bounded_vector w(1,ndim,-10,10)
  vector w1(1,ndim)
  init_vector lambda(1,ndim,2)
  init_vector lambda2(1,ndim,-1)
  init_bounded_vector delta(1,ndim,0,.98)
  sdreport_matrix R(1,ndim,1,ndim)
  sdreport_matrix Omega(1,ndim,1,ndim)
  matrix ch_R(1,ndim,1,ndim)
  matrix Rinv(1,ndim,1,ndim)
  init_bounded_vector v_R(1,ndim2,-1.0,1.0)
  sdreport_matrix Z(1,ndim,1,ndim)
  matrix ch_Z(1,ndim,1,ndim)
  init_bounded_vector v_Z(1,ndim1,-1.0,1.0)
  matrix S(1,ndim,1,ndim);
  objective_function_value f
```

```
INITIALIZATION_SECTION
  delta 0.9
PROCEDURE_SECTION

  fill_the_matrices();
  int sgn;
  ldR=ln_det(R,sgn);
  Rinv=inv(R);
  dvar_vector tmp(1,ndim);
  dvar_matrix sh(1,ndim,1,ndim);
  h_mean(1)=theta0;
  h_var(1)=0;
  for (int i=2;i<=nobs;i++)
  {
    dvar_vector tmean=update_the_means(w,h_mean(i-1),Y(i-1));
    dvar_matrix v=update_the_variances(h_var(i-1));
    tmp=tmean;
    dvar_vector h(1,ndim);
    dvar_vector gr(1,ndim);
    for (int ii=1;ii<=4;ii++)  // do the Newton-Raphson 4 times
    {
      xfp12(tmp, Y(i),tmean,v,gr,sh); // get 1st and 2nd derivatives
      h=-solve(sh,gr);  //sh is hessian and gr is the gradient
      tmp+=h;  // add new step h
    }
    double nh=norm2(value(h)); // check size of h for convergence
    if (nh>1.e-1)
      cout << "No convergence in NR " << nh << endl;
    if (nh>1.e+02)
    {
      f+=1.e+7;   // this ensures that the function minimizer will take a
      return;    // smaller step
    }
    h_mean(i)=tmp;
    h_var(i)=inv(sh);
    lmin(i)=fp(tmp,Y(i),tmean,v);
    int sgn;
    f+=lmin(i)+0.5*ln_det(sh,sgn);  // Laplace approximation
  }
  f-=0.5*nobs*ndim*log(2.*3.14159);
  Omega=S;

FUNCTION  dvar_vector update_the_means(dvar_vector& w,dvar_vector& m,dvector& e)
  dvar_vector tmp= w+elem_prod(delta,m)+elem_prod(lambda,e);
  if (active(lambda2))
    tmp+=elem_prod(lambda2,fabs(e));
  return tmp;

FUNCTION  dvar_matrix update_the_variances(dvar_matrix& v)
  dvar_matrix tmp(1,ndim,1,ndim);
  for (int i=1;i<=ndim;i++)
  {
    for (int j=1;j<=i;j++)
    {
      tmp(i,j)=delta(i)*delta(j)*v(i,j);
      if (i!=j) tmp(j,i)=tmp(i,j);
    }
  }
```

```
  tmp+=Z;
  return tmp;

FUNCTION dvariable fp(dvar_vector& h, dvector& y, dvar_vector& m,dvar_matrix& v)
  dvar_vector eh=exp(.5*h);
  for (int i=1;i<=ndim;i++)
  {
    for (int j=1;j<=i;j++)
    {
      S(i,j)= eh(i)*eh(j)*R(i,j);
      if (i!=j) S(j,i)=S(i,j);
    }
  }

  dvariable lndet;
  dvariable sgn;
  dvar_vector u=solve(S,y,lndet,sgn);
  dvariable l;
  l=.5*lndet+.5*(y*u);
  dvar_vector hm=h-m;
  w1=solve(v,hm,lndet,sgn);
  l+=.5*lndet+.5*(w1*hm);
  return l;

FUNCTION void xfp12(dvar_vector& h, dvector& y,dvar_vector& m,dvar_matrix& v, dvar_vector
gr,dvar_matrix& hess)
  dvar_vector ehinv=exp(-.5*h);
  dvariable lndet;
  dvariable sgn;
  dvar_vector ys=elem_prod(ehinv,y);
  dvar_vector u=Rinv*ys;
  gr=0.5;
  dvar_vector vv=elem_prod(ys,u);
  gr-=.5*vv;
  dvar_vector hm=h-m;
  dvar_vector w=solve(v,hm,lndet,sgn);
  gr+=w;
  for (int i=1;i<=ndim;i++)
  {
    for (int j=1;j<=i;j++)
    {
      hess(i,j)=0.25*ys(i)*ys(j)*Rinv(i,j);
      if (i!=j) hess(j,i)=hess(i,j);
    }
  }
  for (i=1;i<=ndim;i++)
  {
    hess(i,i)+=.25*vv(i);
  }
  hess+=inv(v);

FUNCTION  fill_the_matrices
  int ii=1;
  ch_Z.initialize();
  for (int i=1;i<=ndim;i++)
  {
    for (int j=1;j<=i;j++)
      ch_Z(i,j)=v_Z(ii++);
```

```
    ch_Z(i,i)+=0.5;
  }
  Z=ch_Z*trans(ch_Z);
  ch_R.initialize();
  ii=1;
  for (i=1;i<=ndim;i++)
  {
    for (int j=1;j<i;j++)
      ch_R(i,j)=v_R(ii++);
    ch_R(i,i)+=0.1;
    ch_R(i)/=norm(ch_R(i));
  }
  R=ch_R*trans(ch_R);

REPORT_SECTION
  report<<"observed"<<Y<<endl;
  for (int i=1;i<=nobs;i++)
  {
    report<< "mean" <<endl;
    report<< h_mean(i) <<endl;
    report<< "covariance" <<endl;
    report<<h_var(i)<<endl;
    report<<endl;
  }
  report<< "S(nobs) " << endl;
  report<< Omega << endl;
  report<< "Z " << endl;
  report<< Z << endl;
  report<< "R " << endl;
  report<< R << endl;

TOP_OF_MAIN_SECTION
  arrmblsize=20000000;
  gradient_structure::set_CMPDIF_BUFFER_SIZE(25000000);
  gradient_structure::set_GRADSTACK_BUFFER_SIZE(1000000);
```

Chapter 8

All the functions in AD Model Builder

This chapter attempts to list and document all the functions available in AD Model Builder. It will always be incomplete since functions are continually being added. If you are aware of a function which is not documented please contact me at otter@otter-rsch.com and let me know.

Whereever applicable the name function has been supplied for constand and variable objects (such as `double` and `dvariable`). Instead of repeating the description for bioth kinds of objects the convention of referring to both types as "number", "vector", "matrix", etc. with be observed.

The following functions have been included in AUTODIF by overloading the C++ library functions

```
sin cos tan asin atan acos sinh cosh tanh fabs (sfabs) exp log log10 sqrt pow
gammln log_comb
```

These functions can be used on numbers or vector_objects in the form

```
number = function(number);
vector_object = function(vector_object);
```

When operating on `vector_objects` the functions operate element by element, so that if `y` is a `dvector` whose elements are $(y_1, \ldots, y_n)$ then `exp(y)` is a `dvector` whose elements are $(\exp(y_1), \ldots, \exp(y_n))$.

The functions `min` and `max` when applied to a `vector_object` return a `number` which is equal to the minimum or maximum element of the `vector_object`

The function `gammln` is the logarithm, of the gamma function.

The function `log_comb(n,k)` is the logarithm, of the function the combination of `n` things taken `k` at a time. It is defined via the logarithm of the gamma function for non-integer values and is differentiable.

There are several operations familiar to users of spreadsheets which do not appear as often in classical mathematical calculations. For example spreadsheet users often wish to multiply one column in a spreadsheet by the corresponding elements of another column. Spread sheet users might find it much more natural to define the product of matrices as an element-wise operation such as

$$z_{ij} = x_{ij} * y_{ij}$$

The "classical" mathematical definition for the matrix product has been assigned to the overloaded operator "`*`" so that large mathematical formulas involving vector and matrix operations can be written in a concise notation. Typically, spreadsheet-type calculations are not so complicated and do not suffer so much from being forced to adopt a "function-style" of notation.

Since addition and subtraction are already defined in an element-wise manner, it is only necessary to define element-wise operations for multiplication and division. We have name these functions `elem_prod` and `elem_div`.

```
vector_object = elem_prod(vector_object,vector_object) // element-wise multiply
```

$$z_i = x_i * y_i$$

```
vector_object = elem_div(vector_object,vector_object) // element-wise divide
```

$$z_i = x_i / y_i$$

```
matrix_object = elem_prod(matrix_object,matrix_object) // element-wise multiply
```

$$z_{ij} = x_{ij} * y_{ij}$$

```
matrix_object = elem_div(matrix_object,matrix_object) // element-wise divide
```

$$z_{ij} = x_{ij} / y_{ij}$$

```
matrix_object = identity_matrix(int min,int max)
```

Creates a square identity matrix with minimum valid indices min and maximum valid index max.

The determinant of a matrix object (The matrix must be square, that is the number of row must equal the number of columns)

```
matrix_object = det(matrix_object)
```

The inverse of a matrix object (The matrix must be square, that is the number of row must equal the number of columns)

```
matrix_object = inv(matrix_object)
```

The norm of a vector_object

```
number = norm(vector_object)
```

$$z = \sqrt{\sum_i x_i^2}$$

The norm squared of a vector_object

```
number = norm2(vector_object)
```

$$z = \sum_i x_i^2$$

The norm of a matrix_object

```
number = norm(matrix_object)
```

$$z = \sqrt{\sum_{ij} x_{ij}^2}$$

The norm squared of a matrix_object

```
number = norm2(matrix_object)
```

$$z_{ij} = x_{ji}$$

The transpose of a matrix_object

```
matrix_object = trans(matrix_object)
```

$$z = \sum_{ij} x_{ij}^2$$

The sum over the elements of a vector object

```
number = sum(vector_object)
```

$$z = \sum_i x_i$$

The row sums of a matrix object

```
vector = rowsum(matrix_object)
```

$$z_i = \sum_j x_{ij}$$

The column sums of a matrix object

```
vector = colsum(matrix_object)
```

$$z_j = \sum_i x_{ij}$$

The minimum element of a vector object

```
number = min(vector_object)
```

The maximum element of a vector object

```
number = max(vector_object)
```

While we have included eigenvalue and eigenvector routines for both constant and variable matrix objects you should be aware that in general the eigenvectors and eigenvalues are not differentiable functions of the variables determining the matrix.

The eigenvalues of a symmetric matrix

```
vector_object = eigenvalues(matrix_object)
```

are returned in a vector. It is the users responsibility to ensure that the matrix is actually symmetric. The routine symmetrizes the matrix so that the eigenvalues returned are actually those for the symmetrized matrix.

The eigenvectors of a symmetric matrix

```
matrix_object = eigenvectors(matrix_object)
```

are returned in a matrix. It is the users responsibility to ensure that the matrix is actually symmetric. The routine symmetrizes the matrix so that the eigenvectors returned are actually those for the symmetrized matrix. The eigenvectors are located in the columns of the matrix. The i'th eigenvalue returned by the function `eigenvalues` corresponds to the i'th eigenvector returned by the function `eigenvector`.

For a positive definite symmetric matrix S, the choleski decomposition of S ia a lower triangular matrix T satifying the relationship S=T*trans(T). If S is a (positive definite symmetric) matrix object and T is a matrix object, the line of code

```
T=choleski_decomp(S);
```

will calculate the choleski decomposition of S and put it into T.

If y is a vector and M is an invertible matrix then finding a vector x such that

```
x=inv(M)*y
```

will be referred to as solving the system of linear equations determined by y and M. Of course it is possible to use the inv function to accomplish this task but it is much more efficient to use the solve function.

```
vector x=solve(M,y); // x will satisfy x=inv(M)*y;
```

It turns out that it is a simple matter to calculate the determinant of the matrix M at the same time as the system of linear equations is solved, and since this is useful in multivariate analysis we have also included a function which returns the determinant at the same time as the system of equations is solved. To avoid floating point overflow or underflow when working with large matrices the logarithm of the absolute value of the determinant together with the sign of the determinant are returned The constant form of the solve function is

```
double ln_det;
double sign;
dvector x=solve(M,y,ln_det,sign);
```

while the variable form is

```
dvariable ln_det;
dvariable sign;
dvar_vector x=solve(M,y,ln_det,sign);
```

The solve function is useful for calculating the log-likelihood function for a multivariate normal distribution. Such a log-likelihood function involves a calculation similar to

```
l = -.5*log(det(S)) -.5*y*inv(S)*y
```

where Sis a matrix object and y is a vector object. It is much more efficient to carry out this calculation using the solve function. The following code illustrates the calculations for variable objects.

```
dvariable ln_det;
dvariable sign;
dvariable l;
dvar_vector tmp=solve(M,y,ln_det,sign);
l=-.5*ln_det-y*tmp;
```

While it is always possible to fill vectors and matrices by using loops and filling them element by element, this is tedious and prone to error. To simplify this task a selection of methods for filling vectors and matrices with random numbers or a specified sequence of numbers is available. There are also methods for filling row and columns of matrices with vectors. In this section the symbol `vector` can refer to either a `dvector` or a `dvar_vector`. while the symbol `matrix` can refer to either a `dmatrix` or a `dvar_matrix`.

```
void vector::fill("{m,n,...,}")
```

fills a vector with a sequence of the form `n, m,` ... The number of elements in the string must match the size of the vector.

```
void vector::fill_seqadd(double& base, double& offset)
```

fills a vector with a sequence of the form `base, base+offset, base+2*offset,`...

For example if `v` is a `dvector` created by the statement

```
dvector v(0,4);
```

then the statement

```
v.fill_seqadd(-1,.5);
```

will fill `v` with the numbers $(-1.0, -0.5, 0.0, 0.5, 1.0)$.

```
void matrix::rowfill_seqadd(int& i,double& base, double& offset)
```

fills row `i` of a matrix with a sequence of the form `base, base+offset, base+2*offset,`...

```
void matrix::colfill_seqadd(int& j,double& base, double& offset)
```

fills column `j` of a matrix with a sequence of the form `base, base+offset, base+2*offset,`...

```
void matrix::colfill(int& j,vector&)
```

fills the j'th column of a matrix with a vector

```
void matrix::rowfill(int& i,vector&)
```

fills the i'th row of a matrix with a vector

This mehtiod of filling containers with random numbers is becoming obsolete. the preferred method is to use the `random_number_generator` class. See section 8.15 for instructions on using this class. In this section a uniformly distributed random number is assumed to have a uniform distribution on $[0, 1]$. A normally distributed random number is assumed to have mean 0 and variance 1. A binomially distributed random number is assumed to have a parameter p where 1 is returned with probability p and 0 is returned with probability $1 - p$.

A multinomially distributed random variable is assumed to have a vector of parameters P where i is returned with probability p_i. If the components of P do not sum to 1 the vector will be normalized so that the components do sum to 1.

```
void vector::fill_randu(long int& n)
```

fills a vector with a sequence of uniformly distributed random numbers. The `long int n` is a seed for the random number generator. Changing `n` will produce a different sequence of random numbers. This function is now obsolete. You should use the `random_number_generator` class to generate random numbers.

```
void matrix::colfill_randu(int& j,long int& n)
```

fills column `j` of a matrix with a sequence of uniformly distributed random numbers The `long int n` is a seed for the random number generator. Changing `n` will produce a different sequence of random numbers.

```
void matrix::rowfill_randu(int& i,long int& n)
```

fills row `i` of a matrix with a sequence of uniformly distributed random numbers

```
void vector::fill_randbi(long int& n, double& p)
```

fills a vector with a sequence random numbers from a binomial distribution.

```
void vector::fill_randn(long int& n)
```

fills a vector with a sequence of normally distributed random numbers This function is now obsolete. You should use the `random_number_generator` class to generato random numbers.

```
void matrix::colfill_randn(int& j,long int& n)
```

fills column `j` of a matrix with a sequence of normally distributed random numbers

```
void matrix::rowfill_randn(int& i,long int& n)
```

fills row `i` of a matrix with a sequence of normally distributed random numbers

```
void vector::fill_multinomial(long int& n, dvector& p)
```

fills a vector with a sequence random numbers from a multinomial distribution. The parameter p is a `dvector` such that `p[i]` is the probability of returning i. The elements of `p` must sum to 1.

When this code was first written the maximum dimnsion of arrays was about 4. At this level it perhaps make sense to think of a one dimensional array as a vector, a two dimensional array as a matrix etc. For a matrix one thinks in terms of rows and columns. However with the adoption of ragged container objects up to eight dimensions (at present) a more generic method of obtaining shape information of these objects was called for.

If `v` is a vector object then

```
int v.indexmin()
int v.indexmax()
```

return the minimum and maximum valid indices for v. If `m` is a matrix object then

```
int v.rowmin()
int v.rowmax()
int v.colmin()
int v.colmax()
```

return the minimum and maximum valid row and column indices for m. These functions make sense for a matrix where every row is a vector with the same minimum and maximum valid indices. For a ragged matrix this is no longer the case so that the `rowmin` and `rowmax()` functions don't make sense in this case. To deal with a ragged matrix one may need to calculate the minimum and maximum valid indices for each row of the ragged matrix. To facilitate this approach the functions `indexmin` and `indexmax` have been defined for all container classes so that for example if `w` is a six dimensional array then

```
int w.indexmin()
int w.indexmax()
```

return the minimum and maximum valid indices for the first index of `w`. For a matrix object `m` `m.indexmin()` and `m.colmin()` are the same and as long as `m` is not ragged then `m(m.indexmin()).indexmin()` is the same as `m.colmin()` and `m(m.indexmin()).indexmax()` is the same as `m.colmax()`.

```
vector column(matrix& M,int& j)
```

extracts the j'th column from a matrix and puts it into a vector

```
vector extract_row(matrix& M,int& i)
```

extracts a row from a matrix and puts it into a vector. Note that the operation `M(i)` has the same effect.

```
vector extract_diagonal(matrix& M)
```

extracts the diagonal elements from a matrix and puts them into a vector.

The function call operator `( )` has been overloaded inv two ways to provide for the extraction of a subvector.

```
vector(ivector&)
```

An `ivector` object is used to specify the elements of the vector to be chosen. If `u` and `v` are `dvectors` and `i` is an `ivector` the construction

```
dvector u = v(i);
```

will extract the members of v indexed by i and put them in the `dvector u`. The size of `u` is equal to the size of `i`. The `dvector u` will have minimum valid index and maximum valid index equal to the minimum valid index and maximum valid index of `i`. The size of `i` can be larger than the size of `v` in which case some elements of `v` must be repeated. The elements of the `ivector i` must lie in the valid index range for `v`.

If `v` is a `dvector` and `i1` and `i2` are two integers

```
u(i1,i2)
```

is a `dvector` which is a subvector of `v` (provided of course that `i1` and `i2` are valid indices for `v`). Subvectors can appear on both the left and right hand side of an assignment.

```
dvector  u(1,20);
dvector v(1,19);
v = 2.0;   // assigns the value 2 to all elements of v
u(1,19) = v; // assigns the value 2 to elements 1 through 19 of u
```

In the above example suppose that we wanted to assign the vector `v` to elements 2 through 20 of the vector `u`. To do this we must first ensure that they have the same valid index ranges. The operators `++` and `--` increment and decrement the index ranges by 1. The code fragment

```
dvector  u(1,20);
dvector  w(1,19);
dvector v(1,19);

v = 2.0;   // assigns the value 2 to all elements of v

--u(2,20) = v; // assigns the value 2 to elements 2 through 20 of u

u(2,20) = ++v; // assigns the value 2 to elements 2 through 20 of u
               // probably not what you want
w=v;   // error different index ranges
```

It is important to realize that from the point of view of the vector `u` both of the above assigments have the same effect. It will have elements 2 through 20 set equal to 2. The difference is in the side effects on the vector `v`. The operation `++v` will increase the minimum and maximum valid index range of the vector `v` by one. This increase is permanent. On the other hand the operation `--u(2,20)` decrements the valid index bounds of the *subvector* `u(2,20)`. This is a distinct object from the vector `u` although both objects share a common area for their components. Thus the valid index bounds of `u` are not effected by this process. The use of subvectors and increment and decrement operations can be used to remove loops from the code. Note that

```
dvector x(1,n)
dvector y(1,n)
dvector z(1,n)
for (int i=2;i<=n;i++)
{
  x(i)=y(i-1)*z(i-1);
}
```

can be written as

```
dvector x(1,n)
dvector y(1,n)
dvector z(1,n)
x(2,n)=++elem_prod(y(1,n-1),z(1,n-1));  // elem_prod is element-wise
                                       // multiplication of vectors
```

Ths `shift` function can be sued to set the minimum (and maximum) valid index for a vector.

```
dvector u(10,100);  // minimum valid index is 10
                    // maximum valid index is 100
u.shift(25);        // minimum valid index is 25
                    // maximum valid index is 115
```

In particular the operators `--` and `++` are just convenient shorthand for using the `shift` function to chnage the minimum valid index by 1.

```
dvector u(10,100);     // minimum valid index is 10
                       // maximum valid index is 100
u.shift(u.indexmin()-1); // minimum valid index is 9
--u;                   // same result as u.shift(u.indexmin()-1)
u.shift(u.indexmin()+1); // minimum valid index is 11
++u;                   // same result as u.shift(u.indexmin()+1)
```

While sorting is not strictly a part of methods for calculating the derivatives of differentiable functions (it is a highly non-differentiable operation) it is so useful for pre- and post-processing data that we have included some functions for sorting `dvector` and `dmatrix` objects. If `v` is a `dvector` the statement

```
dvector w=sort(v);
```

will sort the elements of `v` in ascending order and put them in the `dvector` object `w`. The minimum and maximum valid indices of `w` will be the same as those of `v`. If desired an index table for the sort can be constructed by passing and `ivector` along with the `dvector`. This index tables can be used to sort other vectors in the same order as the original vector by using the `()` operator.

```
dvector u={4,2,1};
dvector v={1,6,5}
ivector ind(1,3);
dvector w=sort(u,ind); // ind will contain an index table for the sort
// Now w=(1,2,4) and ind=(3,2,1)
dvector ww=v(ind);     // This is the use of the ( ) operator for subset
                       // selection.
// Now ww=(5,6,1)
```

The sort function for a `dmatrix` object sort the columns of the dmatrix into ascending order, using the column specified to do the sorting. For example

```
dmatrix MM = sort(M,3);
```

will put the sorted matrix into MM and the third column of MM will be sorted in ascending order.

```
cumd_norm
inv_cumd_norm
cumd_cauchy
inv_cumd_cauchy
```

The cumulative distribution function and the inverse cumulative distribution function for the normal and cauchy distributions.

The random number generator class is used to manage the generation of random numbers. A random number generator object is created with the declaration

```
random_number_generator r(n);
```

where n is the seed which initializes the random number generator. Any number of random number generators may be declared. This class can be used to manage random number generation with the folowing functions.

```
randpoisson(lambda,r);  // generate a poisson with parameter lambda
randn(r);     // generate a normally distributed random number
randu(r);    // generate a uniformly distributed random number
v.fill_randu(r)  // fill a vector v
v.fill_randn(r)  // fill a vector v
v.fill_randpoisson(lambda,r)  // fill a vector v
v.fill_multinomial(r,p)  // fill a vector v
                      // p is a vector of probabilities
m.fill_randu(r)  // fill a matrix m
m.fill_randn(r)  // fill matrix m
m.fill_randpoisson(lambda,r)  // fill a matrix m
```

Chapter 9

Advanced Features of AD Model Builder

A useful feature of C++ is its open nature. This means that the user can combine several class libraries into one program. In general this simply involves including the necessary header files in the program and then declaring the appropriate class instances in the program. Instances of external classes can be declared in AD Model Builder program in several ways. They can always be declared in the procedure or report section of the program as local objects. It is sometimes desired to include instances of external classes in a more formal way into an AD Model Builder program. This section describes how to include them into the `DATA_SECTION` or `PARAMETER_SECTION`. After that they can be referred to as though they were part of the AD Model Builder code (except for the technicalities to be discussed below).

AD Model Builder employs a strategy of late initialization of class members. The reason for this is to allow time for the user too carry out any calculations which may be necessary for determining parameter values etc. which are used in the initializatin of the object. Because of the nature of constructors in C++ this means that every object declared in the `DATA_SECTION` or the `PARAMETER_SECTION`must have a default constructor which takes no arguments. The actual allocation of the object is carried out by a class member function named `allocate` which takes any desired arguments. Since external classes will not generally satisfy these requirments a different strategy is employed for these classes. A pointer to the object is included in the appropriate AD Model Builder class. This pointer has the prefix `pad_` inserted before the name of the object. The pointer to `myobj` would have the form `pad_myobj`.

```
!!CLASSfooclass myobj( .... )
```

The user can refer to the object in the code simply by using its name.

The `robust_regression` function calculates the log-likelihood function for the standard statistical model of independent normally distributed errors with mean 0 and equal variance.

The code is written in terms of AUTODIF objects such as dvariable and dvar_vector. They are described in the AUTODIF User's Manual.

```
dvariable regression(const dvector& obs,const dvar_vector& pred)
{
  double nobs=double(size_count(obs));  // get the number of
                                        // observations
  dvariable vhat=norm2(obs-pred);  // sum of squared deviations
  vhat/=nobs;                      //mean of squared deviations
  return (.5*nobs*log(vhat));     //return log-likelihood value
}
```

The effect of a declaration depends on whether it occurs in the DATA_SECTION or in the PARAMETER_SECTION. Objects declared in the DATA_SECTION are constant, that is like data. Objects declared in the PARAMETER_SECTION are variable, that is like the parameters of the model which are to be estimated. Any objects which depend on variable objects must themselves be variables objects, that is they are declared in the PARAMETER_SECTION and not in the DATA_SECTION.

In the DATA_SECTION the prefix init_ indicates that the object is to be read in from the data file. In the PARAMETER_SECTION the prefix indicates that the object is an initial parameter whose value will be used to calculate the value of other (non initial) parameters. In the PARAMETER_SECTION initial parameters will either have their values read in from a parameter file or will be initialized with their default initial values. The actual default values used can be modified in the INITIALIZATION_SECTION. From a mathematical point of view objects declared with the init_ prefix are independent variables which are used to calculate the objective function being minimized.

The prefixes bounded_ and dev_ can only be used in the PARAMETER_SECTION. The prefix bounded_ restricts the numerical values which an object can take on to lie in a specified bounded interval. The prefix dev_ can only be applied to the declaration of vector objects. It has the effect of restricting the sum of the individual components of the vector object to sum to 0.

The prefix sdreport_ can only be used in the PARAMETER_SECTION. An object declared with this prefix will appear in the covariance matrix report. This provides a convenient method for obtaining estimates for the variance of any parameter which may be of interest. Note that the prefixes sdreport_ and init_ can not both be applied to the same object. There is no need to do so since initial parameters are automatically included in the standard deviations report. AD Model Builder also has three and four dimensional arrays. They are declared like

```
3darray dthree(1,10,2,20,3,10)
4darray df(1,10,2,20,3,10)
init_3darray dd(1,10,2,20,3,10)    // data section only
init_4darray dxx(1,10,2,20,3,10)   // data section only
```

The following table contains a summary of declarations and the types of objects assosiciatated with them in AD Model Builder. The types **dvariable, dvector, dmatrix, d3_array, dvar_vector, dvar_matrix, and dvar3_array** are are described in the AUTODIF Users's manual.

declaration	type of object in DATA_SECTION	type of object in PARAMETER_SECTION
[init_]int	int	int
[init_][bounded_]number	double	dvariable
[init_][bounded_][dev_]vector	vector of doubles(dvector)	vector of dvariables(dvar_vector)
[init_][bounded_]matrix	matrix of doubles(dmatrix)	matrix of dvariables(dvar_matrix)
[init_]3darray	3 dimensional array of doubles	3 dimensional array of dvariables
4darray	4 dimensional array of doubles	4 dimensional array of dvariables
5darray	5 dimensional array of doubles	5 dimensional array of dvariables
6darray	6 dimensional array of doubles	6 dimensional array of dvariables
7darray	7 dimensional array of doubles	7 dimensional array of dvariables
sdreport_number	na	dvariable
likeprof_number	na	dvariable
sdreport_vector	na	vector of dvariables(dvar_vector)
sdreport_matrix	na	matrix of dvariables(dvar_matrix)

We have been told that the profile likelihood as calculated in AD Model Builder
for dependent variables may differ from that calculated by other authors. This section will clarify what we mean by the term and motivate our calculation.

Let $(x_1, \ldots, x_n)$ be n independent variables, $f(x_1, \ldots, x_n)$ be a probability distribution and g denote a dependent variable that is a real valued function of $(x_1, \ldots, x_n)$. Fix a value g_0 for g and consider the integral

$$\int_{\{x:g_0-\epsilon/2\leq g(x)\leq g_0+\epsilon/2\}} f(x_1, ..., x_n)$$

which is the probability that $g(x)$ has a value between $g_0 - \epsilon/2$ and $g_0 + \epsilon/2$. This probability depends on two quantities, the value of $f(x)$ and the thickness of the region being integrated over. We approximate $f(x)$ by its maximum value $\hat{x}(g) = \max_{x:g(x)=g_0}\{f(x)\}$. For the thickness we have $g(\hat{x}+h) \approx g(\hat{x}) + < \nabla g(\hat{x}), h >= \epsilon/2$ where h is a vector perpendicular to the level set of g at $\hat{x}$. However ∇g is alos perpendicular to the level set so $< \nabla g(\hat{x}), h >= \|\nabla g(\hat{x})\|\|h\|$ so that $\|h\| = \epsilon/(2\|g(\hat{x})\|)$. Thus the integral is approximated by $\epsilon f(\hat{x})/\|\nabla g(\hat{x})\|$ and taking the derivative with respect to ϵ yields $f(\hat{x})/\|\nabla g(\hat{x})\|$ which is the profile likelihood exprression for a dependent variable.

Bard, Yonathan. Nonlinear Parameter Estimation. Academic Press. N.Y. 1974

Gelman, Andrew., John B. Carlin, Hal S. Stern, and Donald B. Rubin. Bayesian Data Analysis. Chapman and Hall.

Hilborn, Ray and Carl Walters. Quantitative Fisheries Stock Assessment and Management: Choice, Dynamics, and Uncertainty. 1992.

AD Model Builder bundled with AUTODIF is available for a wide variety of compilers on 80386 computers including Borland C++, Visual C++(32 bit) and the "GNU" C++ compiler DJGPP Other compilers are supported at present on Linux, SUN, HP, and SGI UNIX workstations. It is important that you tell us the exact form of your hardware and version of your compiler for UNIX. Multi-user and site licenses are available. Contact

Otter Research Ltd
PO Box 2040,
Sidney, B.C. V8L 3S3
Canada
Voice or Fax (250)-655-3364
Email otter@otter-rsch.com
Internet: otter-rsch.com

Chapter 10

Index